Vorlesung Echtzeitsysteme

Prof. Alois Knoll

Lehrstuhl Embedded Systems and Robotics

Sommersemester 2005

Prof. Alois Knoll

Prof.Dr.-Ing. Alois Knoll
Sekretariat Frau M. Knürr, Frau R. te Vehne
Raum: 03.07.054
Email: knoll@in.tum.de

Dipl.-Inf. Christian Buckl
Raum: 03.07.061
Email: buckl@in.tum.de

Vorlesung

3 SWS Vorlesung im Bereich Informatik II (Technische Informatik)
2 SWS Übung

Wahlpflichtvorlesung im Gebiet Echtzeitsysteme
Wahlpflichvorlesung für Studenten der Elektro- und Informationstechnik
Pflichtvorlesung für Studenten Maschinenbau Richtung Mechatronik

Schein: Soweit im Einzelfall ein Schein benötigt wird, ist die Basis dafür eine 30-minütige Prüfung.

Skript: Folien zur Vorlesung im Netz auf den Lehrstuhlseiten.

Vorlesungszeiten

Vorlesung:

- Donnerstags 10:15-11:45 Uhr MI 00.06.011 (HS3)
- Freitags 11:15-12:00 Uhr MW 0001 (Maschinenwesen)

Übung: zweistündig
Montags 13:30-18:00 Uhr
Raum: 03.05.012

Anmeldung zur Übung

Teilnehmerzahlen an Übung begrenzt
Anmeldung über https://grundstudium.in.tum.de

Zur Anmeldung ist ein Zertifikat der Rechnerbetriebsgruppe nötig. Nähere Informationen sind unter http://ca.in.tum.de/userca/ zu finden.

Studenten, die nicht am Rechenbetrieb teilnehmen, erhalten ihr Zertifikat unter http://grundstudium.in.tum.de/zertifikat.

Weitere Angebote des Lehrstuhls

- Vorlesungen: Robotik, Maschinelles Lernen und bioinspirierte Optimierung I & II, Sensor- und kamerageführte Roboter
- Praktika:Echtzeitsysteme, Roboterfußball, Roboter, Neuronale Netze und Maschinelles Lernen
- Seminare:Sensorik, Objekterkennung und Lernen, Neurocomputing, Monitoring und Regelung in der Medizin
- Diplomarbeiten
- Masterarbeiten
- Systementwicklungsprojekte

Literatur

- Alans Burns, Andy Wellings: Real-Time Systems and Programming Languages
- Stuart Bennet: Real-Time Computer Control: An Introduction
- Bruce Powel Douglass: Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns
- Bill O. Gallmeister: Programming for the Real-World: POSIX.4

Weitere Literaturangaben befinden sich in den jeweiligen Abschnittten.

Vorlesungsinhalte

- Einführung Echtzeitsysteme
- Fehlertoleranz
- Modellierung
- Werkzeuge
- Betriebssysteme
- Programmiersprachen
- Uhren
- Nebenläufigkeit
- Scheduling
- Kommunikation
- Spezielle Hardware
- Regelungstechnik

Weitere Themen können bei Interesse aufgenommen werden (Umfrage im Anschluss an die heutige Vorlesung).

Einführung

- Definition Echtzeitsysteme
- Klassifikation
- Echtzeitsysteme im täglichen Einsatz
- Beispielanwendungen am Lehrstuhl

ca. 1 Vorlesungsstunde

Fehlertoleranz

- Bekannte Softwarefehler
- Definitionen
- Fehlerarten
- Fehlerhypothesen
- Fehlervermeidung
- Fehlertoleranzmechanismen

ca. 1 Vorlesungsstunden

Übung: Design und Implementierung von Fehlertoleranzmechanismen

Modellierung

- Allgemein
- Esterel
- TLA+
- Lustre
- Zerberus System

ca. 2 Vorlesungsstunden

Übung: Modellierung von Applikationen in verschiedenen Modellen

Werkzeuge

- Rhapsody
- Artisan
- Zerberus System

ca. 1 Vorlesungsstunde

Echtzeitbetriebssysteme

- QNX
- RTLinux, RTAI
- Linux Kernel 2.6
- PineOS
- TinyOS
- eCos
- OSEK
- Beck-IPC
- VxWorks (Vortrag Windriver)

ca. 0,5 + 1 (Vortrag Windriver) Vorlesungsstunden

Überblick Kapitel Programmiersprachen

- Ada
- Posix.4
- Real-Time Java

ca.0,5 Vorlesungsstunden

Übung: C-, Ada-Programmierung

Uhren

- Uhren
- Synchronisation von verteilten Uhren

ca.0,5 Vorlesungsstunden

Übung: Uhrensynchronisation im Mehrrechnersystem

Nebenläufigkeit

- Prozesse und Threads
- Interprozesskommunikation

ca. 0,5 Vorlesungsstunden

Übung: Thread-Programmierung, Realisierung unterschiedlicher IPC

Scheduling

- Kriterien
- Planung Einrechner-System, Mehrrechnersysteme
- EDF, Least Slack Time
- Scheduling mit Prioritäten (FIFO, Round Robin)
- Scheduling periodischer Prozesse
- Scheduling Probleme

ca. 1 Vorlesungsstunde

Übung: Implementierung eines EDF-Schedulers

Echtzeitfähige Kommunikation

- Token-Ring
- CAN-Bus
- TTP
- FlexRay
- Real-Time Ethernet

ca. 1 Vorlesungsstunde

Übung: CAN-Bus–Programmierung

Spezielle Hardware

- Digital-Analog-Converter (DAC)
- Analog-Digital-Converter (ADC)
- Speicherprogrammierbare Steuerung (SPS)

ca. 1 Vorlesungsstunde

Regelungstechnik

- Definitionen
- P-Regler
- PI-Regler
- PID-Regler
- Fuzzy-Logic

ca. 1 Vorlesungsstunde

Übung: Regelung der “schwebenden Jungfrau”

Einführung Echtzeitsysteme

Inhalt

- Definition Echtzeitsysteme
- Klassifikation von Echtzeitsystemen
- Echtzeitsysteme im täglichen Leben
- Beispielanwendungen am Lehrstuhl

Definition Echtzeitsystem

Ein **Echtzeit-Computersystem** ist ein Computersystem, in dem die Korrektheit des Systems nicht nur vom logischen Ergebnis der Berechnung abhängt, sondern auch vom physikalischen Moment, in dem das Ergebnis produziert wird.

Ein **Echtzeit-Computer-System** ist immer nur ein Teil eines größeren Systems, dieses größere System wird **Echtzeit-System** genannt.

Hermann Kopetz
TU Wien

Resultierende Anforderungen

- Zeitliche Genauigkeit
- Garantierte Antwortzeiten

Aber nicht:

- Allgemeine Geschwindigkeit

Zeitlicher Determinismus vs. Leistung

Implikationen der Forderung nach determisitischer Ausführungszeit: Mechanismen, die die allgemeine Performance steigern, aber einen negativen, nicht exakt vorhersehbaren Effekt auf einzelne Prozesse haben können, dürfen nicht verwendet werden:

- Virtual Memory
- Garbage Collection
- Asynchrone IO-Zugriffe
- rekursive Funktionsaufrufe

Klassen von Echtzeitsystemen

Strenge der Zeitkritischen Anforderungen:

- hart
- weich

Ausführungsmodell:

- Zeitgesteuert (zyklisch, periodisch)
- Ereignisgesteuert (aperiodisch)

Harte bzw. weiche Echtzeitsysteme

Weiche Echtzeitsysteme:
Eingehende Aufgaben haben eine vorgegebene Reaktionszeit, deren Nicht-Beachtung jedoch noch nicht sofort katastrophale Auswirkungen hat.

Harte Echtzeitsysteme: Eine Nicht-Beachtung der vorgegebenen Reaktionszeit führt sofort zum maximalen Schaden führen. Die Einhaltung der Zeitvorgaben ist strikt.

Anforderungen je nach Ausführungsmodell

Zeitgesteuerte Applikationen:

- Präzise, konstante Uhr, evtl. Uhrensynchronisation
- Einhaltung der vorgeschriebenen Fristen (worst case execution times)
- Statisches Scheduling (→vorhersagbares Verhalten)

Ereignisgesteuerte Applikationen:

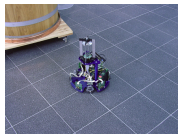
- Garantierte Anwortzeiten
- Dynamisches Scheduling

Echtzeitsysteme im Alltag

Echtzeitsysteme im Alltag

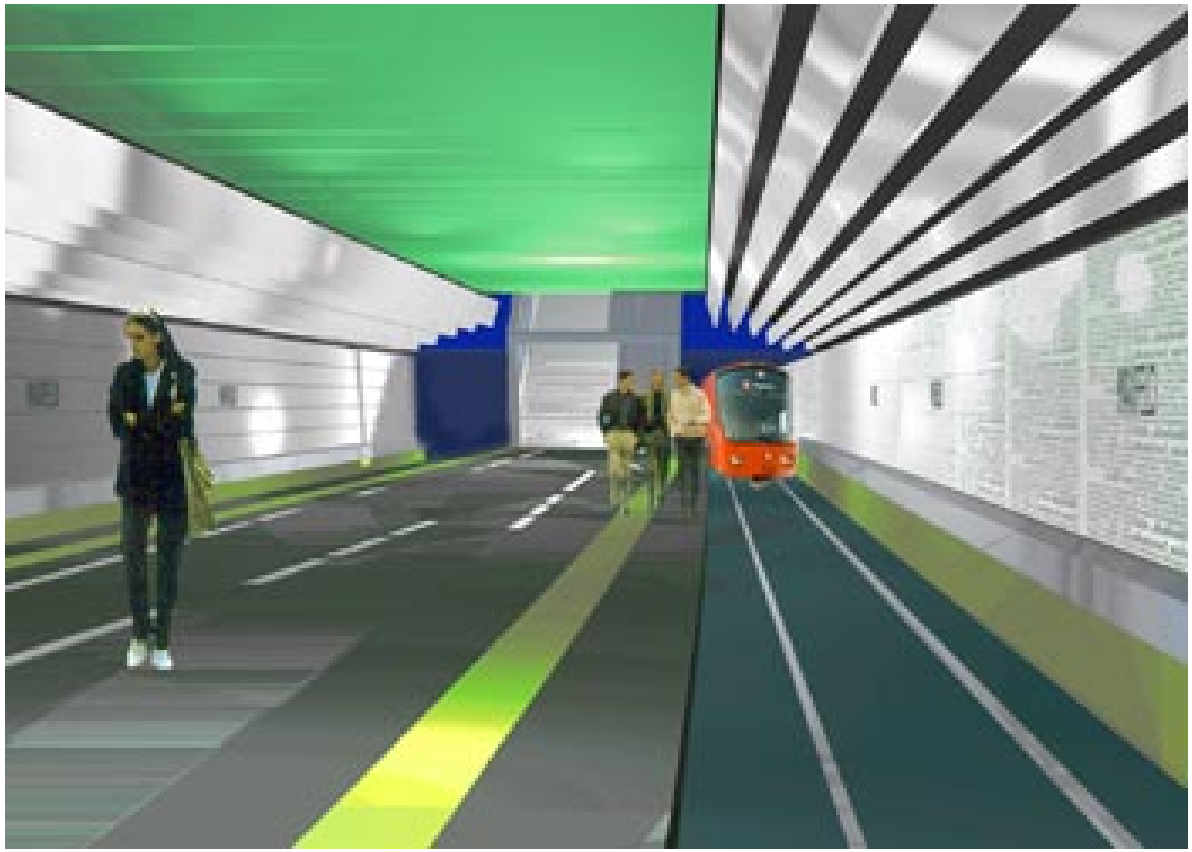




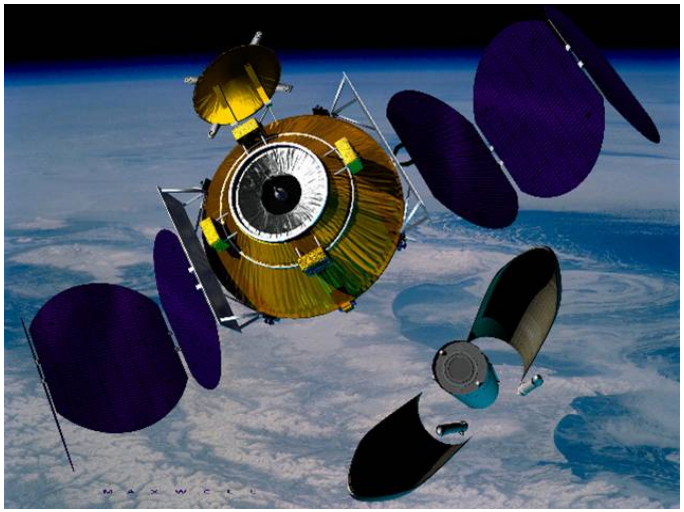

Echtzeitsysteme sind allgegenwärtig!

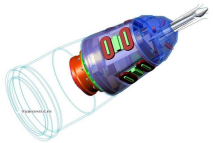
U-Bahn Nürnberg

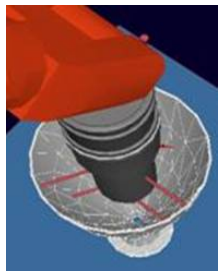


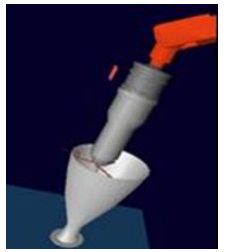
Vollautomatisierte Steuerung einer U-Bahn

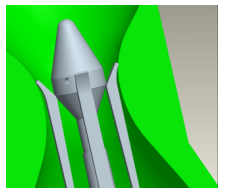
CxOlev









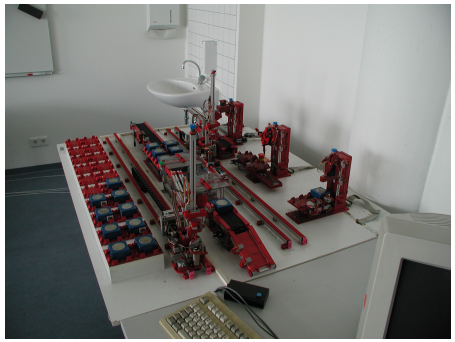


Lebenszeitverlängerung von Satelliten

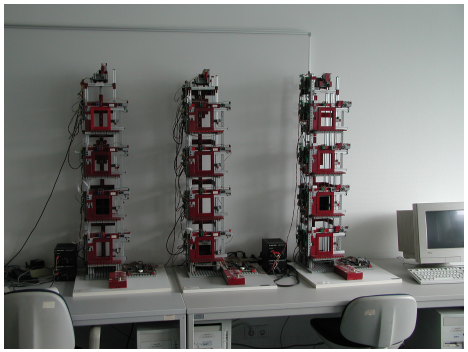
Anwendungen am Lehrstuhl

Steuerungsaufgaben

Modelleisenbahn



Industrieanlage



Aufzugsteuerung

Regelungsaufgaben

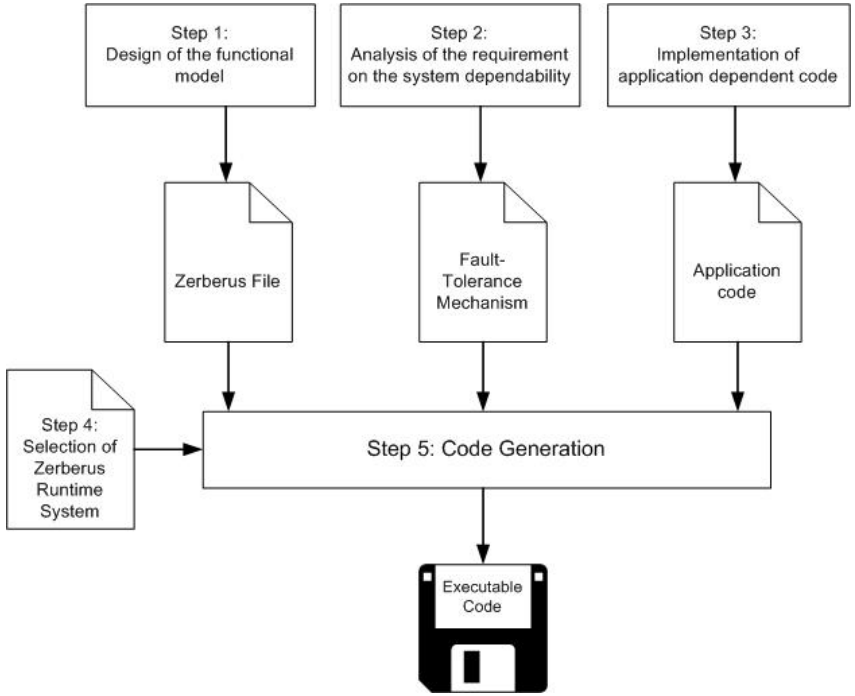
Schwebende Jungfrau

CaVaCo - Automatische
Regelung der Medikamentengabe

Modellierung von sicheren und zuverlässigen Echtzeitsystemen

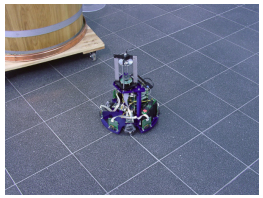
Zerberus System

- Toolgestütztes Entwicklungsmodell
- Basierend auf Mehrrechnersystem
- Minimiert den Entwicklungsaufwand

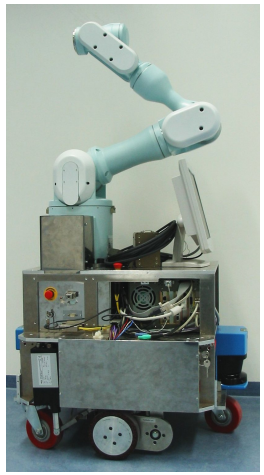


Robotersteuerung

Khepera



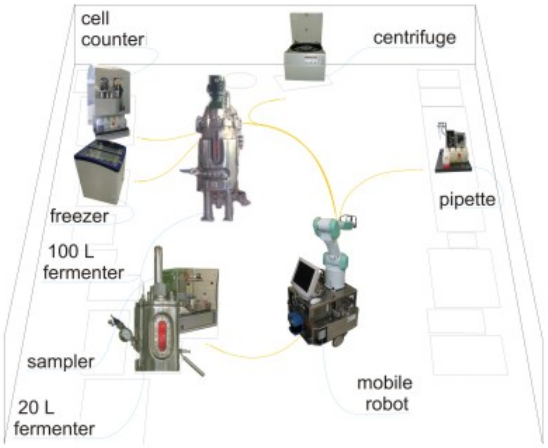
Robotino



Leonardo

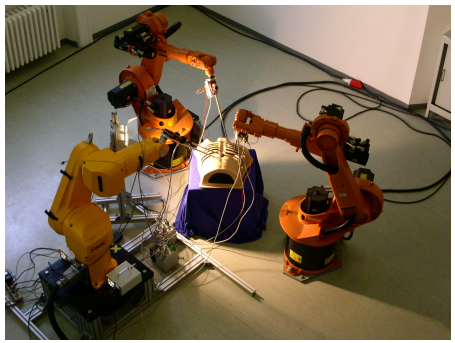
Stäubli

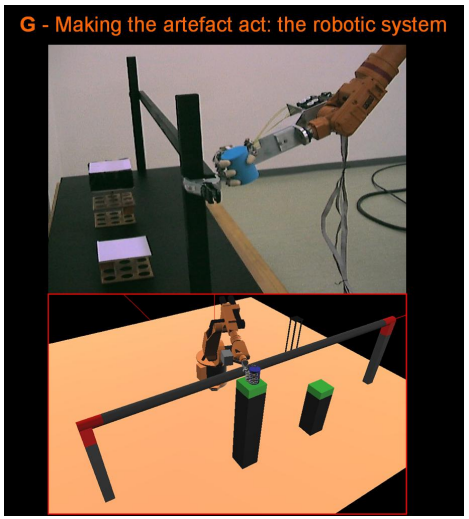
Anwendungen der Robotik



Automation von biotechnologischen Laboren

Telemedizin





ArteSImit

Vorlesung Echtzeitsysteme - Fehlertoleranz

Prof. Alois Knoll

Lehrstuhl Embedded Systems and Robotics

Sommersemester 2005

Inhalt

- Einleitung
- Grundlagen
- Fehlertoleranzmechanismen
- Quantitative Bewertung fehlertoleranter Systeme

Literatur

- Dhiraj K. Pradhan: Fault-Tolerant Computer System Design, Prentice Hall 1996
- Klaus Echtle: Fehlertoleranzverfahren, Springer-Verlag 1990 (elektronisch unter http://dc.informatik.uni-essen.de/Echtle/all/buch_ftv/)
- W.A.Halang, R.Konakovsky: Sicherheitsgerichtete Echtzeitsysteme, Oldenburg 1999
- Peter G.Neumann: Computer Related Risks, ACM Press 1995
- Nancy G.Leveson: Safeware, Addison-Wesley 1995

- http://www.system-safety.org/

Ariane 5 (1996)

Selbstzerstörung bei Jungfernflug:

Design:

- 2 redundante Meßsysteme (identische Hardware und Software) bestimmen die Lage der Rakete (hot-standby)
- 3-fach redundante On-Board Computer (OBC) überwachen Meßsysteme

Ablauf:

- Beide Meßsysteme schalten aufgrund eines identischen Fehlers ab
- OBC leitet Selbstzerstörung ein

Ursache:

- Wiederverwendung von nicht-kompatiblen Komponenten der Ariane 4.

Weitere Informationen unter
http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.

Therac 25 (1985-1987)

Computergesteuerter Elektronenbeschleuniger zur Strahlentherapie

Das System beinhaltete 3 schwere Mängel:

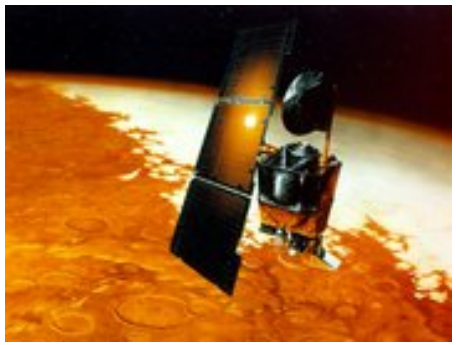
- Sicherheitsprüfungen im Programm wurden durch einen Softwarefehler bei jeder 64. Benutzung ausgelassen (wenn ein 6-bit Zähler Null wurde).
- Behandlungsanweisungen konnten mittels Editieren am Bildschirm so teilgeändert werden, daß die Maschine für die nächste Behandlung nicht den gewünschten Zustand einnahm (nämlich Niederintensität).
- Mehrere Sicherheitsverriegelungen, die beim Vorgängermodell Therac-20 in Hardware realisiert waren, wurden nicht übernommen, sondern durch Software ersetzt.

Folgen:

- Mehrere Patienten erhielten anstatt der vorgesehenen Dosis von 80-200 rad Strahlungsdosen von bis zu 25000 rad (mehrere Tote und Schwerverletzte).

Weitere Informationen unter http://sunnyday.mit.edu/papers/therac.pdf.

Mars Climate Orbiter (1998)



Verglühen beim Eintritt in die Atmosphäre

Ursache:

- Verwendung von unterschiedlichen Maßeinheiten (Zoll, cm) bei der Implementierung der einzelnen Komponenten.
- Mangelnde Erfahrung, Überlastung und schlechte Zusammenarbeit der Bodenmannschaften

Weitere Informationen unter
http://mars.jpl.nasa.gov/msp98/orbiter/.

Explosion Chemiefabrik (1992)

Explosion einer holländischen Chemiefabrik aufgrund eines Bedienfehlers

- Computergesteuertes Mischen von Chemikalien.
- Operateur (in Ausbildung) verwechselt beim Eintippen eines Rezeptes 632 (Harz) mit 634 (Dicyclopentadien).
- Explosion fordert 3 Menschenleben, Explosionsteile finden sich noch im Umkreis von 1 km.

Grundlagen

- Definitionen

Anforderungen an Systeme

Systeme zum Einsatz in sicherheitskritischen Anwendungen erfordern ein hohes Maß an **Systemstabilität (dependability)**. Dieser Begriff umfasst:

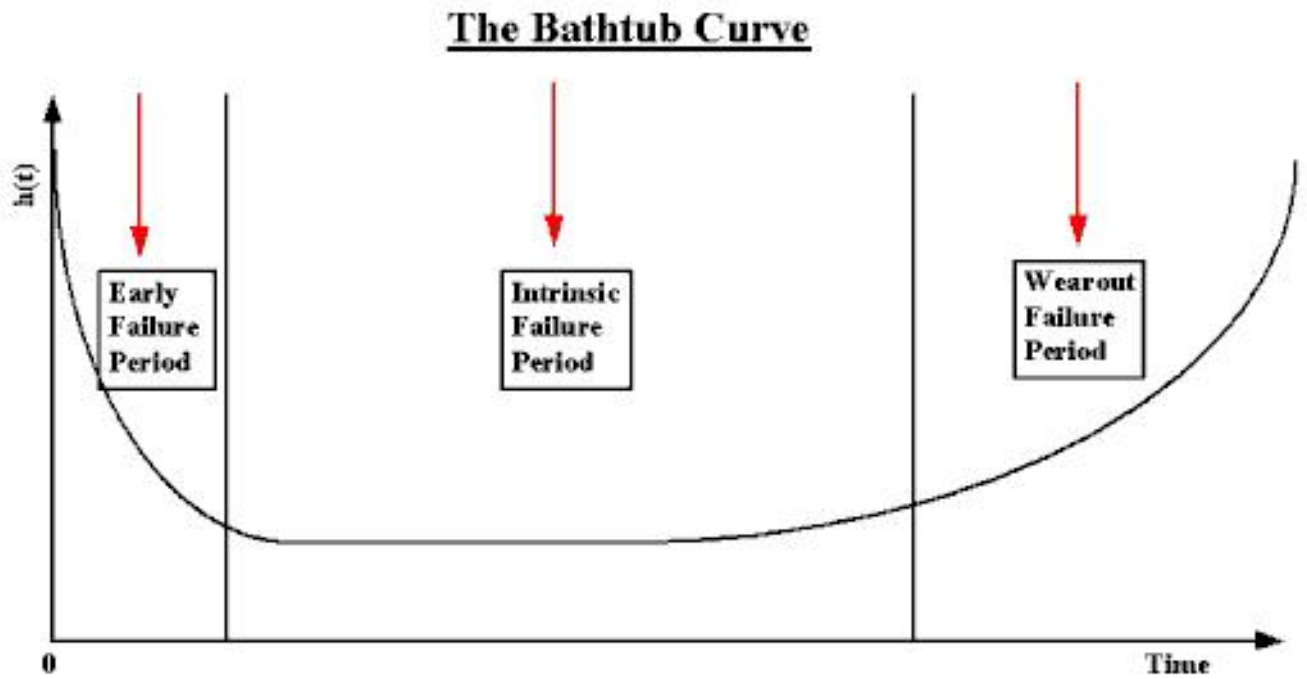
- Zuverlässigkeit
- Sicherheit
- Verfügbarkeit
- Leistungsfähigkeit
- Robustheit
- Wartbarkeit
- Testbarkeit

Fehlerrate

Die **Fehlerrate** gibt die erwartete Anzahl an Fehler eines Gerätes oder eines Systems für eine gegebene Zeitperiode an.

Typischerweise wird die Fehlerrate als konstant angenommen (siehe Badewannenkurve) und mit λ bezeichnet. Typische Einheit der Fehlerrate ist Fehler pro Stunde.

Fehlerrate in Abhängigkeit von der Zeit (Badewannenkurve)



Zuverlässigkeit

Definition:
Die **Zuverlässigkeit (reliability)** eines Systems ist eine Funktion $0 \leq R(t) \leq 1$, definiert als die bedingte Wahrscheinlichkeit, dass das System korrekt während des Intervals $[t_0, t]$ funktioniert unter der Annahme, dass das System zum Zeitpunkt t_0 korrekt arbeitete.

Wird eine konstante Fehlerrate angenommen, so kann die Zuverlässigkeit durch folgende Gleichung angegeben werden:

$$R(t) = e^{-\lambda(t-t_0)}$$

Frage:
Was ist die korrekte Funktionalität eines Systems?

Sicherheit

Definition:
Sicherheit(safety) ist die Wahrscheinlichkeit $0 \leq S(t) \leq 1$, dass ein System entweder korrekt arbeitet oder seine Funktion auf eine Art und Weise beendet, so dass es nicht die Funktionsweise anderer Systeme gestört oder Menschen gefährdet werden. Sicherheit ist damit ein Maßstab für die Fähigkeit eines Systems auf eine sichere Art und Weise auszufallen.

Frage: Wie kann ein System sicher ausfallen (**fail-safe**)?

Verfügbarkeit

Definition:
Verfügbarkeit (availability) wird als eine Funktion $0 \leq A(t) \leq 1$ über die Zeit ausgedrückt, die die Wahrscheinlichkeit angibt, dass ein System zum Zeitpunkt t korrekt arbeitet. Im Gegensatz zur **Zuverlässigkeit** wird bei der **Verfügbarkeit** neben der Häufigkeit der Dienstausfälle auch die Dauer der Reparaturen und Wartungsarbeiten berücksichtigt.

Während bei der Zuverlässigkeit die Korrektheit des Systems zu allen Zeitpunkten eines gegebenen Intervalls gefordert wird, gibt die Verfügbarkeit die momentane Wahrscheinlichkeit der korrekten Ausführung des Systems an. Eine hohe Verfügbarkeit ist beispielsweise bei transaktionsbasierten Systemen z.B. ein Fluglinienreservierungssystem nötig. Wartungsarbeiten und Reparaturen sollten schnell durchgeführt werden, eine andauernde korrekte Funktion im Sinne der Zuverlässigkeit wird hingegen nicht gefordert.

Leistungsfähigkeit

Definition:
In vielen Fällen ist es möglich und sinnvoll Systeme zu konstruieren, die nach Auftreten von Hardware oder Softwarefehler in einzelnen Komponenten (siehe spätere Einführung von Fehlerbereichen) in einem degradierten Modus weiterarbeiten.
Unter **Leistungsfähigkeit (performability)** wird eine Funktion $0 \leq P(L, t) \leq 1$ über der Zeit verstanden, die eine Wahrscheinlichkeit angibt, dass die Funktionalität des Systems zum Zeitpunkt t mindestens das Niveau L erreicht. Im Gegensatz zur **Zuverlässigkeit**, bei der immer nur die Wahrscheinlichkeit angegeben wird, dass alle Funktionen korrekt funktionieren, können nun auch Teilmengen betrachtet werden.

Robustheit

Definition:
Unter **Robustheit (robustness)** eines Systems wird die Fähigkeit verstanden auch unter erschwerten Betriebsbedingungen (z.B. Fehleingaben (siehe Chemiefabrik) oder widersprüchlichen Meßwerten) die korrekte Funktionalität zu wahren.

Wartbarkeit

Definition:
Wartbarkeit (maintainability) ist ein Maßstab für die Reparaturfreundlichkeit eines Systems. Quantitativ kann die Wartbarkeit als die Wahrscheinlichkeit M(t) ausgedrückt werden, dass das fehlerhafte System innerhalb einer Zeitdauer t repariert werden kann.

Testbarkeit

Definition:
Testbarkeit (testability) ist ein Maßstab für die Möglichkeit bestimmte Eigenschaften eines Systems zu testen.
So kann es möglich sein, bestimmte Tests zu automatisieren und als Mechanismen in das System zu integrieren.

Die Testbarkeit eines Systems ist durch die hohe Bedeutung der schnellen Fehleranalyse direkt mit der Wartbarkeit eines Systems verbunden.

Sicherheitsklassen IEC 62304 CD

Class A	No injury may occur to the patient or to the operator resulting from a hazard to which the software item may be a contributing fator.
Class B	Non-serious injury may occur to the patient or the operator resulting from a hazard to which the software item may be a contributing factor.
Class C	Death or serious injury may occur to the patient or the operator resulting from a hazard to which the software item may be a contributing factor.

Sicherheitsklassen für Medizinprodukte nach IEC

Sicherheitsklassen RTCA/DO-178B

Catastrophic	Failure conditions which would prevent continued safe flight and landing.
Hazardous/Severe-Major	Failure conditions which would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions to the extent that there would be a large reduction in safety margins or functional capabilities, physical distress or higher workload such that the flight crew could not be relied to perform their tasks accurately or completely, or adverse effects on occupants including serious or potentially fatal injuries to a small number of those occupants.
Major	Failure conditions which would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions to the extent that there would be, for example, a significant reduction in safety margins or functional capabilities, a significant increase in crew workload or in conditions impairing crew efficiency, or discomfort to occupants, possibly including injury.
Minor	Failure conditions which would not significantly reduce aircraft safety, and which would involve crew actions that are well within their capabilities. Minor failure conditions may include, for example, a slight reduction in safety margins or functional capabilities, a slight increase in crew workload, such as, routine flight plan changes, or some inconveniance to occupants.
No effect	Failure conditions which do not affect the operational capability of the aircraft or increase crew workload.

Sicherheitsklassen in der Flugzeugindustrie nach RTCA

Konzepte zur Erhöhung der Systemstabilität

Systemstabilität (Definition Folie 8,umfassender Begriff für Zuverlässigkeit, Sicherheit, Verfügbarkeit...) kann durch Anwendung der folgenden Konzepte erreicht werden:

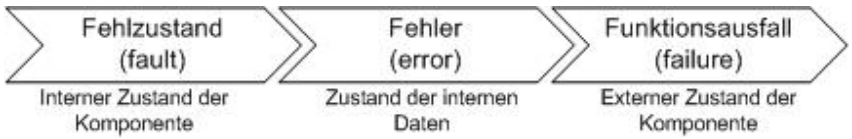
- Fehlervermeidung
  - Designmethoden + Werkzeugunterstützung
  - Modellierung mit Anwendung von Verifikations- und Validierungsmethoden
- Fehlerentfernung
  - Einheitentests
  - Integrationstests
  - Back-To-Back Testing (Vergleich von Resultaten unterschiedlicher Versionen bei N-Versionsprogrammierung)
- Fehlertoleranz

Fehlertoleranzmechanismen

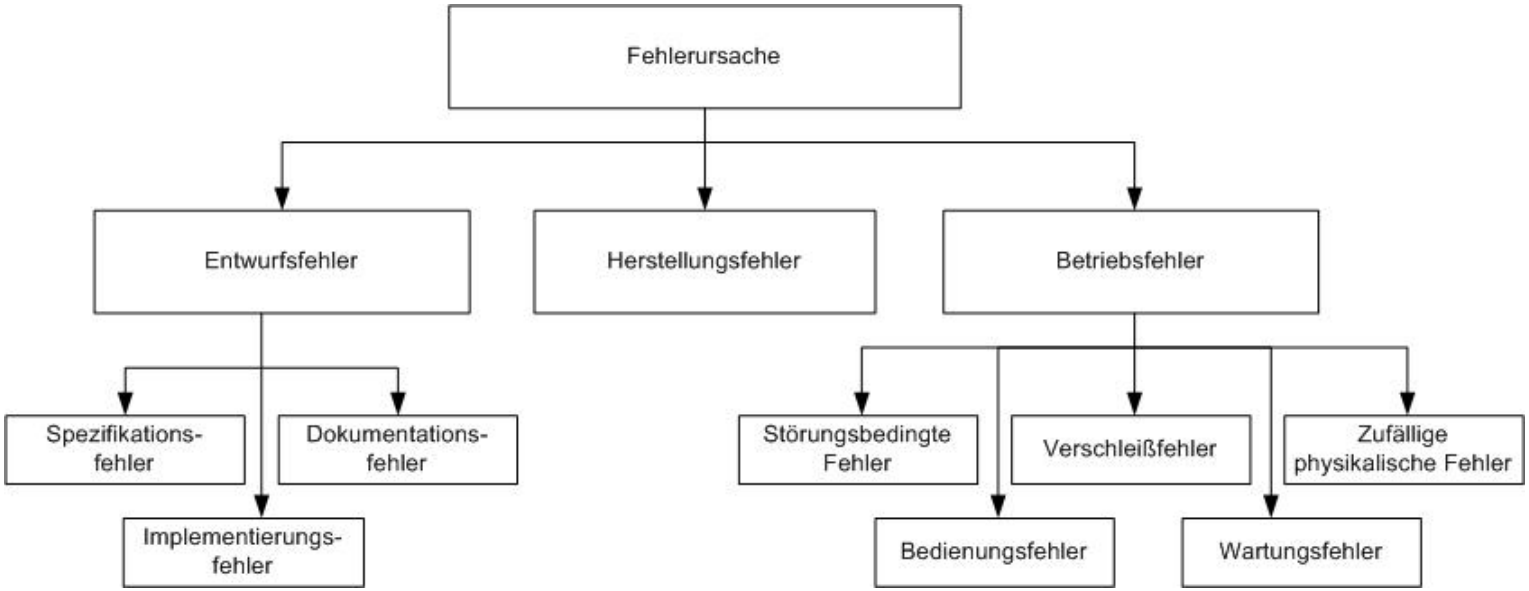
- Fehlermodell

Begriffe:

- Fehlzustand(fault): physikalischer Fehler oder Störstelle in einer Hardware- oder Softwarekomponente
- Fehler (error): Erscheinungsform eines Fehlzustands, z.B. durch das Abweichen eines Wertes vom erwarteten Wert in den internen Daten
- Funktionsausfall (failure): Ausfall oder fehlerhafte Durchführung von Funktionen eines Systems, Auftritt an der Benutzerschnittstelle



Fehlerursachen



Klassifizierung von Fehlern

Unterscheidung nach Entstehungsort:

- Hardware
- Software

Unterscheidung nach Fehlerdauer:

- permanent
- intermittierend (flüchtig)
  - periodisch
  - wiederkehrend
  - einmalig

Fehlermodell

Um die Fehlertoleranz-Fähigkeit eines Rechensystems spezifizieren zu können, ist eine **Fehlervorgabe** erforderlich, welche die Menge der zu tolerierenden Fehler auf ein formales **Fehlermodell** angibt.
Ein **Fehlermodell** hat den Zweck zu jedem Zeitpunkt die Fehlermöglichkeiten eines Systems als eine Obermenge der **Menge der zu tolerierenden Fehler** anzugeben.
Das **Fehlermodell** beinhaltet daher

- die Komponenten, die von Fehlern betroffen sein können (**strukturelle Fehlerbetrachtung**) und
- in welcher Art und Weise deren Funktion beeinträchtigt wird (**funktionelle Fehlerbetrachtung**)

Fehlerbereich

Typischerweise wird angenommen, dass Fehler nur in bestimmten Teilmengen der Menge aller Komponenten S auftreten. Jede dieser Komponentenmengen wird als **Fehlerbereich** Fb bezeichnet.

Die Annahmen

- $Fb_1 \cup ... \cup Fb_n \neq S \rightarrow$ es gibt einen Perfektionskern $S \backslash (Fb_1 \cup ... \cup Fb_n)$
- $\exists i, j \in \{1...n\} : Fb_i \cap Fb_j \neq \emptyset \rightarrow$ Überschneidungen sind erlaubt

sind zulässig.

k-Fehler-Annahme

Da die Anzahl der Fehlerbereiche mitunter sehr groß werden kann, bietet sich als Spezialfall der Fehlerbereichsannahme die **k-Fehler-Annahme** an.
Grundlage hierfür ist die disjunkte Zerlegung eines Systems S in Einzelfehlerbereiche $Eb_1, ..., Eb_m$ mit $Eb_1 \cup ... \cup Eb_m = S$. Die k-Fehlerannahme fordert die Tolerierung von allen Fehlern, die sich auf bis zu k Einzelfehlerbereiche erstrecken.
Die bei k-Fehler-Annahme mit $k \geq 2$ zu tolerierenden Fehlerfälle werden **Mehrfachfehler** genannt. Es wird jedoch nicht zwischen zufälligen und systematischen Mehrfachfehlern unterschieden. Dieser Unterschied muss jedoch bei der Anfälligkeitsanalyse genau betrachtet werden.

Beispiel: 3-Rechner-System, als Einzelfehlerbereiche werden die einzelnen Rechner angesehen

Fehlfunktionsannahmen

Detaillierung der Fehlervorgabe durch **Fehlfunktionsannahme**. Sinnvolle Annahmen sind:

- Teil-Ausfall: nur manche Funktionen eines Systems fallen aus, die übrigen werden korrekt erbracht
- Unterlassungs-Ausfall: es wird entweder ein richtiges oder gar kein Ergebnis ausgegeben (ommision fault)
- Anhalte-Ausfall: sobald ein Fehler aufgetreten ist, gibt das System kein Ergebnis mehr aus (fail-stop) $\rightarrow$ jedes ausgegebenen Ergebniss ist korrekt und es fehlt kein früheres Ergebnis
- Haft-Ausfall: ab Auftreten eines Fehlers wird immer das gleiche Ergebnis ausgegeben
- Inkonsistenz-Ausfall: ausgegebene fehlerhafte Ergebnisse sind in sich nicht konsistent (z.B. CRC)
- Binärstellen-Ausfall (oder k-Binärstellenausfall): Fehler verfälschen maximal k Binärstellen eines Ergebnisses
- Nicht-Angriffs-Ausfall: z.B. Schutz von fehlerfreien Komponenten for falscher Authentifikation fehlerhafter Komponenten

Fehlerausbreitung und -eingrenzung

Fehler breiten sich in der Regel ohne geeignete Maßnahmen innerhalb eines Systems aus. Fehlertoleranzverfahren basieren jedoch zumeist auf einer eingeschränkten Fehlervorgabe. So kann zumeist nur eine begrenzte Anzahl an fehlerhaften Komponenten toleriert werden.

$\longrightarrow$ Eingrenzungsmaßnahmen müssen getroffen werden

Typischerweise werden deshalb Maßnahmen zur Isolierung getroffen:

- Hardwarekomponenten werden räumlich getrennt oder gekapselt.
- Software wird so strukturiert, dass möglichst viele Berechnungen in einzelnen Modulen erfolgt.
- An Schnittstellen werden Inkonsistenzprüfungen zwischen den einzelnen Komponenten durchgeführt.

Fehlertoleranzmechanismen

- Redundanz

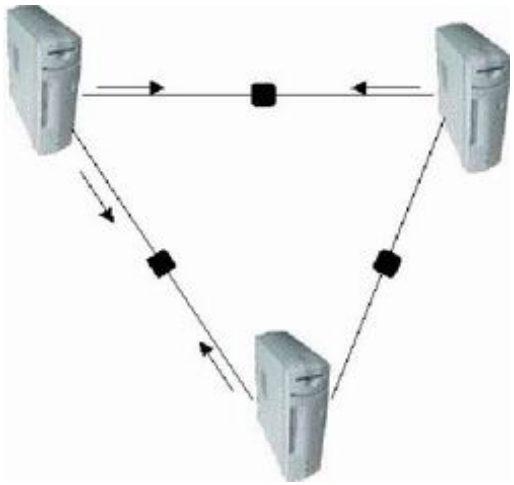
Grundlage der Fehlertoleranzmechanismen: Redundanz

Die beiden grundsätzlichen Schritte eines Fehlertoleranzverfahrens, die Diagnose und Behandlung von Fehlern, benötigen zusätzliche Mittel, die über die Erfordernisse des Nutzbetriebs hinausreichen. All diese zusätzlichen Mittel sind unter dem Begriff **Redundanz** zusammengefasst.

Redundanz bezeichnet das funktionsbereite Vorhandensein von mehr technischen Mitteln, als für die spezifizierte Nutzfunktion eines Systems benötigt werden.

Typische Ausprägung der Redundanz: 2-von-3 System

Ein 2-von-3 System / TMR-System (triple modular redundacy) besteht aus 3 gleichwertigen Komponenten. Ein Ausfall einer Komponente kann toleriert werden, ohne dass die Funktion beeinflusst wird. Bei einem Ausfall einer zweiten Komponente muss in einen sicheren Modus geschaltet werden.



→ Betriebmodi:

- sicherer und zuverlässiger Betrieb (2-von-3-Betrieb)
- sicherer Betrieb (2-von-2-Betrieb)

Zuverlässigkeit redundanter Systeme

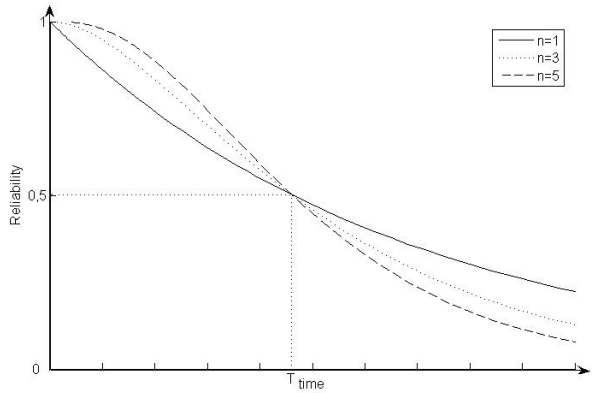
Redundanz kann, muss aber nicht die Zuverlässigkeit verbessern:

Beispiel: 2-von-3 System, stochastisch unabhängige Fehler, konstante Ausfallsrate λ, R_1: Zuverlässigkeit einer Komponente, R_3: Zuverlässigkeit des Systems

$$\rightarrow R_3(t) = R_1(t)^3 + 3 * R_1(t)^2 * (1 - R_1(t)) \tag{1}$$

Allgemeiner Fall m-von-n System:

$$\rightarrow R_n(t) = \sum_{k=m}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) R_1^k * (1 - R_1)^{n-k} \tag{2}$$



$\rightarrow$ ohne Möglichkeiten zur Reparatur sinkt die Zuverlässigkeit eines redundanten Systems nach einer Zeitdauer T unter die Zuverlässigkeit eines einfach ausgelegten Systems

Redundanzarten

Redundanz ist möglich in:

- Hardware (strukturelle Redundanz)
- Information
- Zeit
- Software (funktionelle Redundanz)
  - Zusatzfunktion
  - Diversität

→ Fehlertolerante Rechensysteme setzen zumeist Kombinationen verschiedener redundanter Mittel ein.

Strukturelle Redundanz

Strukturelle Redundanz bezeichnet die Erweiterung eines Systems um zusätzliche (gleich- oder andersartige) für den Nutzbetrieb entbehrliche Komponenten.

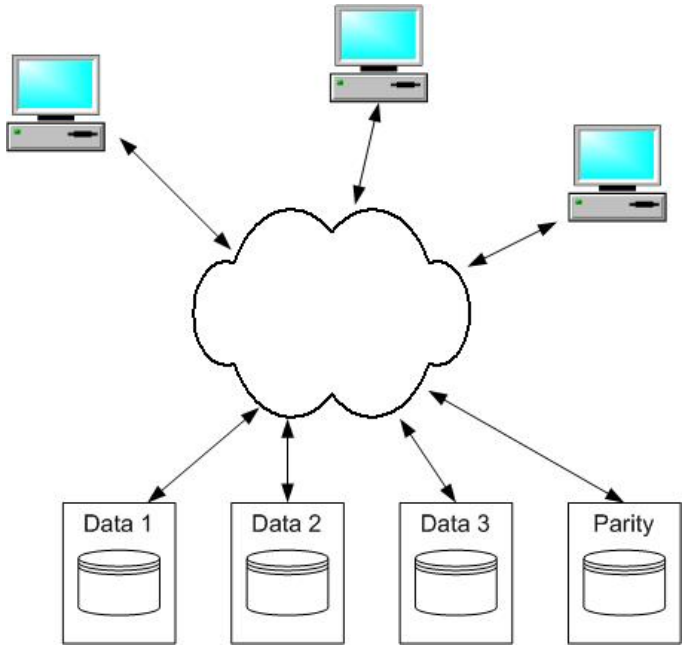
Beispiele:

- 2-von-3-Rechnersysteme
- mehrfache Kopien einer Datei

Anwendungsbeispiel: Verteilte Dateisysteme NetRAID (Lübeck), xFs (Berkeley)

System: verteiltes Speichersystem
Ziel:

- skalierbare Speichergröße
- höhere Zugriffsraten
- Ausfallstoleranz



Funktionelle Redundanz

Funktionelle Redundanz bezeichnet die Erweiterung eines Systems um zusätzlich für den Nutzbetrieb entbehrliche Funktionen.

Beispiele:

- Testfunktionen
- Rekonfigurierungsfunktionen im Mehrrechnerbetrieb
- Erzeugung eines Paritätsbits

Diversität (N-Versions-Programmierung)

Diversität bezeichnet die Erfüllung der Spezifikation einer Nutzbetriebs-Funktion durch mehrere verschiedenartig implementierte Funktionen.

Um Entwurfsfehler in Hard- und / oder Softwaresystemen tolerieren zu können, ist der Einsatz von Diversität zwingend. Diversität verbessert die Zuverlässigkeit aber nicht unbegrenzt. Die Verbesserungsgrenze ist insbesondere von der Schwierigkeit des zu lösenden Systems vorgegeben. Um Diversität zu realisieren, muss der Entwurfsspielraum für die verschiedenen Varianten genutzt werden.

Ansätze:

- Unabhängiger Entwurf
- Gegensätzlicher Entwurf

Beispiel aus dem Alltag: Arztbesuch

Typisches Beispiel für N-Versionsprogrammierung: Konsultation von verschiedenen Ärzten

- Unterschiedliche Spezialisierungen
- Unterschiedliche Untersuchungen
- Unterschiedliche Behandlungsmethoden

Informationsredundanz

Informationsredundanz bezeichnet zusätzliche Informationen neben der Nutzinformation.

Beispiele:

- Fehlerkorrigierende Codes
- Doppelt verkettete Listen

Voraussetzung: Fehler dürfen sich nur auf einen beschränkten Teil der gesamten Information auswirken (z.B. Fehlfunktions-Annahme)

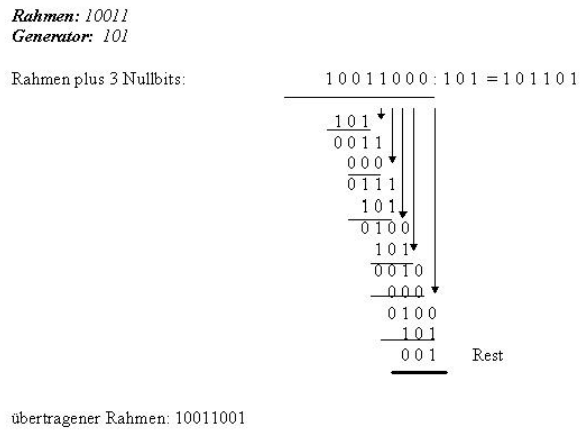
Beispiel: CRC

Cyclic Redundancy Check: typisches Verfahren in Rechnernetzprotokollen zur Entdeckung von Fehlern.

Funktion:

- Interpretation einer Bitfolge (des sog. Rahmens) als Koeffizienten eines Polynoms (z. B. 10011 für $x^4 + x + 1$) M(x),
- Anhängen von Nullbits an der niederwertigen Seite des Rahmens gemäß der Gradzahl des Generatorpolynoms G(x), (z. B. 3 für das Generatorpolynom zu 101, d. h. G(x) = $x^2 + 1$),
- Division des Rahmens durch das Generatorpolynom nach der Modulo-2-Division, die keinen Übertrag berücksichtigt,
- Anhängen des Resultats an den Rahmen anstelle der zuvor angehängten Nullbits.

Typisches Generatorpolynom: $x^{16} + x^{15} + x^2 + 1$



Fragen:

- Wieso werden selten fehlerkorrigierende Codes in Rechnernetzprotokollen eingesetzt?
- Wieso werden keine Parity Bits eingesetzt?

Zeitredundanz

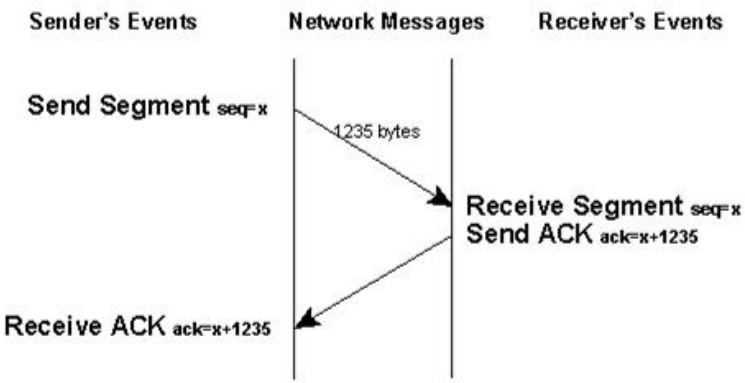
Zeitredundanz bezeichnet über den Zeitbedarf des Normalbetriebs hinausgehende zusätzliche Zeit, die einem funktionell redundanten System zur Funktionsausführung zur Verfügung steht.

Beispiele:

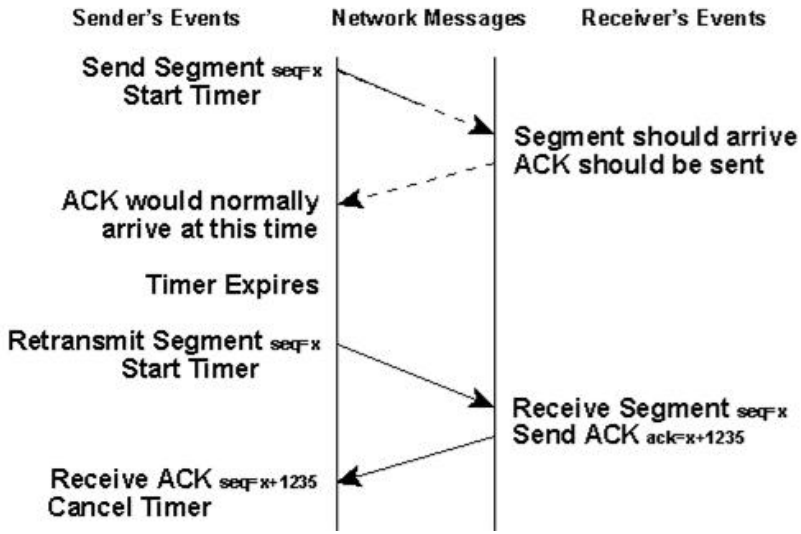
- Wiederholungsbetrieb
- Zeitbedarf für Konsistenzmechanismen in verteilten Dateisystemen

Beispiel: TCP

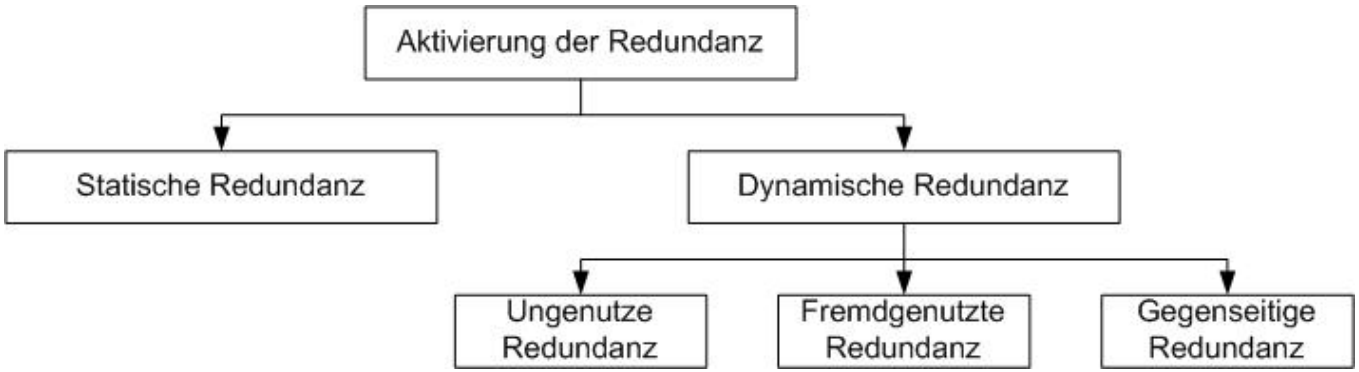
Fehlerfrei



Verlust einer Nachricht



Aktivierung der Redundanz



Statische Redundanz

Statische Redundanz bezeichnet das Vorhandensein von redundanten Mitteln, die während des gesamten Einsatzzeitraums aktiv zu den zu unterstützenden Funktionen beitragen.

Ausprägungen:

- Statische strukturelle Redundanz: z.B. n-von-m System
- Statische funktionelle Redundanz (Zusatzfunktionen): z.B. doppeltes Senden von Nachrichten auf unterschiedlichen Wegen
- Statische funktionelle Redundanz (Diversität): N-Versions-Programmierung
- Statische Informationsredundanz: fehlerkorrigierende Codes
- Statische Zeitredundanz: statische Mehrfachausführung einer Funktion

Dynamische Redundanz

Dynamische Redundanz bezeichnet das Vorhandensein von redundanten Mitteln, die erst im Ausnahmebetrieb (d.h. nach Auftreten eines Fehlers) aktiviert werden, um zu den zu unterstützdenen Funktionen beizutragen.

Typisch für dynamisch strukturelle Redundanz ist die Unterscheidung in **Primärkomponenten** und **Ersatzkomponenten**. Die Dauer der Umschaltung hängt im wesentlichen von den ggf. erforderlichen Vorbereitungsmaßnahmen der Ersatzkomponenten ab. Hier wird zwischen **heißer Reserve (hot standby)** und **kalter Reserve (cold standby)** unterschieden.

Die Definition verlangt kein vollkommen passives Verhalten. Folgende Szenarien sind möglich:

- ungenutzte Redundanz: Ersatzkomponenten sind bis zur fehlerbedingten Aktivierung passiv
- fremdgenutzte Redundanz: Ersatzkomponenten erbringen nur Funktionen, die von den zu unterstützenden Funktionen verschieden sind und im Fehlerfall storniert werden
- gegenseitige Redundanz: Komponenten stehen sich gegenseitig als Reserve zur Verfügung. Im Fehlerfall übernimmt eine Komponente die Funktionen der anderen zusätzlich zu den eigenen.

Fehlertoleranzmechanismen

- Fehlererkennung

Fehlererkennung

Grundlage der Fehlertoleranzmechanismen ist die Fehlerdiagnose. Ziele der Fehlerdiagnose ist:

- das Erkennen von Fehlern (im Nutzbetrieb)
- die Lokalisierung von Fehlern (zumeist im Ausnahmebetrieb)
- die Bestimmung des Behandlungsbereichs (zumeist im Ausnahmebetrieb)

Fehlererkennung

Möglichkeiten zur Fehlererkennung:

- Zeitschrankenüberwachung
- Absoluttests: getestet wird direkt das Ergebnis (z.B. Anzahl der Elemente muss nach Sortieren gleich der eingegebenen Anzahl sein)
- Relativtests: Vergleich von mehreren Ergebnissen redundanter Prozesse
  - bei deterministischen Prozessen
  - bei indeterministischen Prozessen
- Nutzung von Informationsredundanz (z.B. CRC)

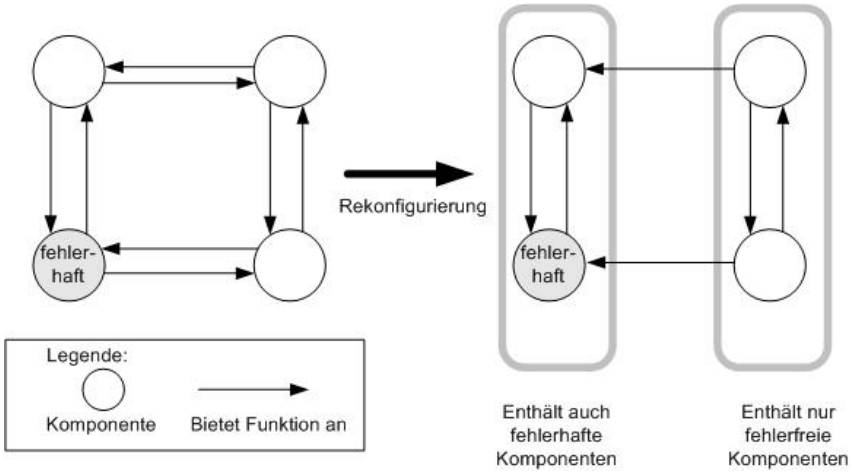
Fehlertoleranzmechanismen

- Mechanismen

Rekonfigurierung

Durch **Rekonfigurierung** werden fehlerhafte Komponenten ausgegrenzt und bestehende Funktionszuordnungen zwischen fehlerhaften und fehlerfreien Komponenten aufgelöst.

Nach einer Rekonfigurierung ist das System in zwei Komponenten-Teilmengen partitioniert: eine enthält nur fehlerfreie, die andere auch fehlerhafte Komponenten.



Rekonfigurierung: Beitrag zur Fehlertoleranz

Rekonfigurierung dient zur Behandlung von Funktionsausfällen, nicht aber der Behebung von Fehlzuständen.

$\rightarrow$ nicht ausreichend für erfolgreiche Fehlerbehandlung
$\rightarrow$ Verfahren zur Fehlerbehebung (Rückwärts-, Vorwärtsbehebung) oder Fehlerkompensierung (Fehlermaskierung, Fehlerkorrektur) müssen hinzukommen.

Rückwärtsbehebung (backward error recovery)

Rückwärtsbehebung versetzt Komponenten in einen Zustand, den sie bereits in der Vergangenheit angenommen hatten oder als konsistenten Zustand hätten annehmen können.
„Konsistent“bedeutet, dass die lokalen Komponentenzustände und die aktuellen Interaktionen mit anderen Komponenten die (Protokoll- bzw. Dienst-)Spezifikation nicht verletzen.

Rückwärtsbehebung ist bei intermittierenden Fehlern aureichend, bei permanenten Fehlern ist sie als Ergänzung zur Rekonfigurierung zu sehen.

Bei reiner Rückwärtsbehebung kann die Fehlererkennung nur über Absoluttests (da keine redundanten Ergebnisse vorhanden) erfolgen. Diese Tests werden periodisch oder ereignisabhängig durchgeführt.

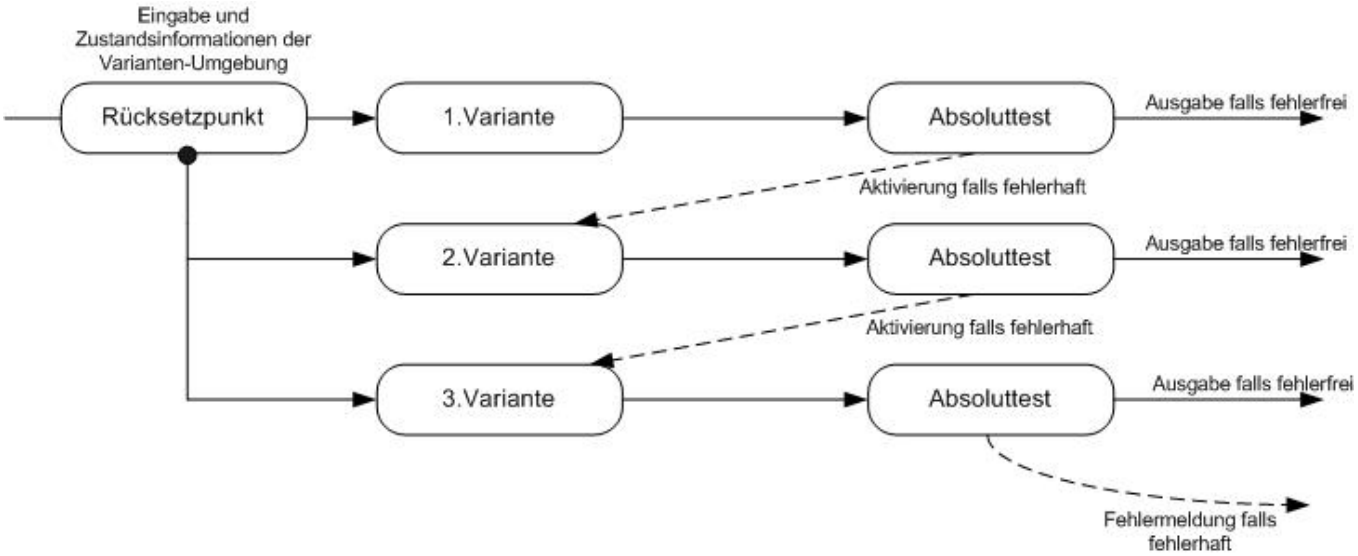
Rücksetzpunkte (recovery point)

Nach Auftreten eines Fehlers lassen sich Zustandsinformationen aus der Zeit vor Auftreten eines Fehlers nur gewinnen, wenn die Informationen zuvor kopiert wurden und an einem getrennten Ort zwischengespeichert wurden. Die abgespeicherte Zustandsinformation wird als Rücksetzpunkt bezeichnet.

Rücksetzpunkte werden periodisch oder ereignisbasiert erstellt und verursachen also schon im Normalbetrieb einen Extrazeitaufwand. Zumeist finden vor der Rücksetzpunkterstellung Absoluttests statt.

Auch Rücksetzpunkte können fehlerhaft sein (Auftreten eines Fehlers direkt nach Absoluttest bzw. beim Speichern des Rücksetzpunktes, Fehlererkennung mit einer Wahrscheinlichkeit <1) . Deshalb muss das System eventuell mehrfach zurückgesetzt werden.

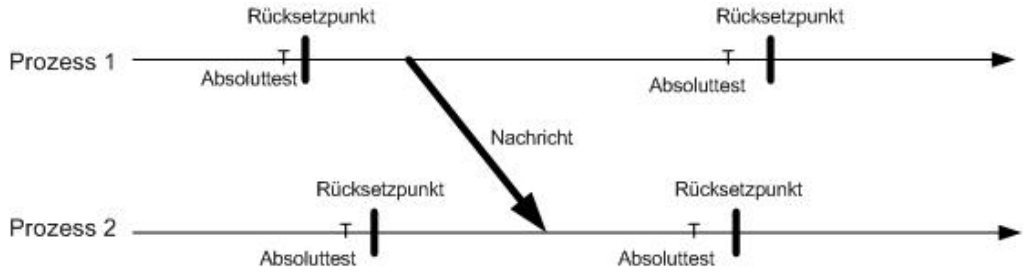
Rückwärtsbehebung diversitärer Systeme



Dieses Verfahren entspricht dem Ausprobieren mehrerer Funktionen. Z.B. das Testen von verschiedenen Browsern (bis Benutzer ein Programm gefällt, die Anzeige einer Internetseite korrekt ist).

Rücksetzlinien bei interagierenden Prozessen

Betrachten wir folgendes Beispiel:



Tritt bei einem der beiden Prozesse zwischen den jeweiligen Rücksetzpunkten ein Fehler auf, so müssen beide Prozesse zurückgesetzt werden.
Warum?

Definition: Die Menge der Rücksetzpunkte, auf die mehrere Prozesse zugleich zurückgesetzt werden können, heißt **Rücksetzlinie (recovery line)**.

Mögliche Probleme: Dominoeffekt

Praktische Fragen

Typische Fragen bei der Implementierung von Rückwärtsbehebung:

- Wo werden die Zustandsinformationen abgespeichert?
- Wann und wie häufig werden Rücksetzpunkte erstellt?
- Wieviele Rücksetzpunkte sind zugleich aufzubewahren?
- Welche Variablen sollen gespeichert werden?
- Wie realisiert man Rücksetzlinien bei interagierenden Prozessen?

Vor- und Nachteile der Rückwärtsbehebung

- Rückwärtsbehebung verwendet die vorhandenen Betriebsmittel sparsam.
- Im Fehlerfall lässt sich ein Prozess wiederholt zurücksetzen (solange Rücksetzpunkte vorhanden), dadurch erhöht sich die Menge der tolerierenden Fehler.
- Wiederholungsbetrieb erfordert nicht zwangsläufig die gleichen Eingaben wie der zuvor erfolgte Nutzbetrieb (Indeterminismus zulässig).
- Rückwärtsbehebung ist transparent (unabhängig von der Anwendung) implementierbar.
- Nur Absoluttests, keine Relativtests anwendbar.
- Menge der tatsächlichen tolerierten Fehler ist wegen der Abhängigkeit von verschiedenen Absoluttest-Algorithmen kaum formal spezifizierbar.
- Rücksetzpunkterstellung kostet schon im Normalbetrieb.
- Der im Fehlerfall erforderliche Wiederholungsbetrieb kann Zeitredundanz in beträchtlichem Umfang fordern.

Vorwärtsbehebung

Vorwärtsbehebung bezeichnet Fehlerbehebungs-Verfahren, die keine Zustandsinformationen der Vergangenheit verwenden.

Basis dieser Verfahren sind Fehlfunktions-Annahmen und anwendungsspezifisches Wissen.

Geht aufgrund eines Fehlers in einem Rechner ein zuvor gelesener Temperaturwert verloren, so kann er durch zweimaliges Einlesen der aktuellen Temperatur und Extrapolation näherungsweise zurückgewonnen werden (Voraussetzung: zeitliche Ableitung der Temperatur ändert sich kaum).

Vor- und Nachteile der Vorwärtsbehebung

- Aufwand an struktureller Redundanz ist gering: nur Absoluttests und die erst im Ausnahmebetrieb zu aktivierenden Ausnahmebehandler sind hinzuzufügen
- Laufzeitaufwand im Normalbetrieb wird nur von Absoluttests verursacht und ist daher minimal
- Vorwärtsbehebung ist nicht transparent sondern anwendungsabhängig
  - hoher Entwurfsaufwand
  - Gelingen hängt stark vom Schwierigkeitsgrad der Anwendung ab
- Nur durch Absoluttests erkennbare Fehler sind überhaupt tolerierbar
- Oft nur degradierter Betrieb nach Vorwärtsbehebung möglich

Fehlermaskierung

Das Verfahren der **Fehlermaskierung** berechnet aus redundant berechneten Ergebnissen ein korrektes Ergebnis zur Weitergabe.

Typischerweise ist die “Maske “durch einen Mehrheitsentscheider realisiert, der Ergebnisse durch einen Relativtest vergleicht. Dieses Verfahren toleriert fehlerhafte Ergebnisse solange diese in der Minderheit bleiben.

Typische Ausprägungen sind 2-von-3-Systeme oder 3-von-5 Systeme.

Maskierungsentscheidungen/Voting

für **deterministische** Prozesse:

- Mehrheitsentscheidung: Mehrheit der Gesamtanzahl von Komponenten nötig
- Paarentscheidung: Annahme, dass fehlerhafte Ergebnisse nie gleich sind → zwei übereinstimmende Ergebnisse sind immer korrekt (Reduzierung der Anzahl nötiger Vergleiche)
- Meiststimmenentscheidung
- Einstimmigkeitsentscheidung: alle Komponenten müssen im Ergebnis übereinstimmen (sehr sicheres, aber nicht zuverlässiges Verfahren)

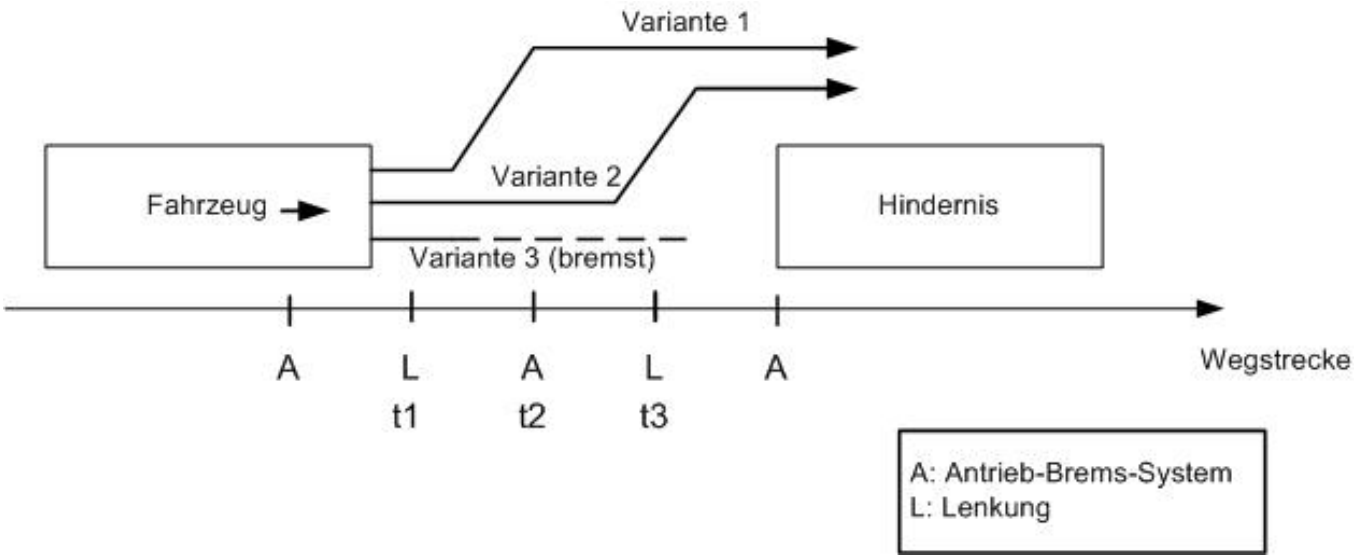
für **indeterministische** Prozesse:

- Medianentscheidung: Mittleres Ergebnis wird für die Ausgabe übernommen
- Intervallentscheidung: Annahme, dass korrekte Ergebnisse in einem Intervall liegen, ein Ergebniswert aus Intervall wird gewählt
- Kugelentscheidung: wie Intervallentscheidung nur mit mehrdimensionalen Ergebnissen, statt Intervall wird nach kürzesten Abständen gesucht

Vor- und Nachteile der Fehlermaskierung

- Fehlermaskierung reicht als **einziges Fehlertoleranz-Verfahren** aus.
- Maskierer lassen sich vergleichsweise einfach implementieren.
- Wiederholungsbetrieb entfällt, dadurch ist die Fehlerbehandlung schneller.
- Fehlerhafte Subsystemexemplare dürfen beliebiges fehlerhaftes Verhalten zeigen, da Relativtests angewandt werden.
- Fehlermaskierung ist transparent zu implementieren.
- Hoher Aufwand durch strukturelle Redundanz

Probleme bei Fehlermaskierung diversitärer Systeme



Probleme bei Fehlermaskierung diversitärer Systeme

Das System berechnet zu unterschiedlichen Zeitpunkten neue Werte für Antrieb (Bremsen, Beschleunigen, konstant fahren) und Lenkung (gerade, links, rechts).

Zum Zeitpunkt t1 wird ein neuer Wert für die Lenkung berechnet: alle Varianten entscheiden sich für geradeaus fahren.
Zum Zeitpunkt t2 wird ein neuer Wert für den Antrieb berechnet: die Mehrheit (Variante 1 und 2) entscheiden sich für konstant fahren.
Zum Zeitpunkt t3 wird ein neuer Wert für die Lenkung berechnet: die Mehrheit (Variante 1 und 3) entscheidet sich für geradeaus fahren.

$\longrightarrow$ es kommt zur Kollision

Forderung: Varianten, die überstimmt wurden, müssen für eine bestimmte Zeit ausgeschlossen werden.

Synchronisation von redundanten Einheiten

Alternativen:

- Gesteuerte Synchronisierung: Steuerung von zentraler Stelle
- Geregelte Synchronisierung: Synchronisation durch Maskierer
- Implizite Synchronisierung: anwendbar falls die Auftragsrate die Bearbeitungsrate stets unterschreitet oder Ergebnisse verschiedener Aufträge vergleichbar sind

Reparatur und Integration von redundanten Einheiten

Wie bereits gesehen sinkt die Zuverlässigkeit eines redundanten Systems nach einer bestimmten Zeitdauer immer unter die Zuverlässigkeit eines einfach ausgelegten Systems falls keine Reparaturen möglich sind. Zuverlässige Systeme mit langen geplanten Betriebszeiten müssen deshalb Reparaturen unterstützen.
Ablauf:

- Erkennen eines Fehlers
- Ausgliedern der fehlerhaften Einheit
- Zeitunkritische Durchführung von Fehlerdetektions- und Fehlerbehebungsalgorithmen
- Wiedereingliederung (Integration)

Wie kann sich eine Einheit wieder integrieren (state synchronisation) ohne den normalen Betrieb zu stören?

Fehlerkorrektur

Fehlerkorrektur bildet einzelne fehlerhafte Ergebnisse, die genügend Informationsredundanz enthalten auf fehlerfreie ab.

Basis ist eine Einschränkung in der Fehlfunktionsannahme (zumeist k-Binärstellen-Ausfall)

$\longrightarrow$ Anwendungsbereich vor allem, wo physikalische Gesetze diese Annahme rechtfertigen:

- bei der Übertragung und
- bei der Speicherung von Daten

Vor- und Nachteile der Fehlerkorrektur

- Strukturelle Redundanz und Zeitredundanz werden nur im geringen Umfang benötigt. Die erforderliche Informationsredundanz ist im Allgemeinen mit geringen Aufwand zu erzeugen und zu überprüfen.
- Bezüglich der Fehlervorgabe weist der Absoluttest eine hohe Fehlererfassung auf.
- Fehlerkorrektur lässt bei der Fehlervorgabe keine beliebigen Ergebnisverfälschungen zu.
- Im Allgemeinen kein geeignetes Mittel um Entwurfsfehler zu korrigieren.

Quantitative Bewertung

Quantitative Bewertung

Typische Kenngrößen fehlertoleranter Systeme:

- Fehlerrate
- Zuverlässigkeit
- MTTF (Mean time to failure)
- MTTR (Mean time to repair)
- MTBF (Mean time between failures)

Modelle zur Berechnung der Zuverlässigkeit:

- Kombinatorische Modelle
- Markow-Modelle

MTTF

MTTF (Mean time to failure) gibt die erwartete Zeitdauer an bis der erste Fehler in dem System auftritt.
Die MTTF kann experimentell bestimmt werden, indem N identische Systeme zum Zeitpunkt $t = 0$ aktiviert werden und die durchschnittliche Zeitspanne bis zum Auftritt eines Fehlers berechnet wird. Falls jedes System i bis zum Zeitpunkt t_i arbeitet, so ist MTTF durch

$$MTTF = \sum_{i=1}^{N} \frac{t_i}{N}$$

gegeben.
Ist die Zuverlässigkeit $R(t)$ des Systems gegeben, so kann MTTF auch direkt berechnet werden:

$$MTTF = \int_0^\infty R(t)dt$$

MTTR

Mit **MTTR (mean time to repair)** wird die durchschnittliche Zeitdauer angegeben, die nötig ist um das System zu reparieren. Die MTTR ist sehr schwierig zu bestimmen und wird zumeist durch experimentelles Einspeisen von Fehlern und dem Messen der Zeitdauer für die Reparatur des Systems bestimmt.
Wird für das Reparieren des i.ten von N Fehlern die Zeit t_i benötigt, so wird MTTR als

$$MTTR = \sum_{i=1}^{N} \frac{t_i}{N}$$

geschätzt.
Typischerweise wird MTTR durch eine Reparaturrate μ, die die Anzahl der Reparaturen pro Zeiteinheit angibt, spezifiziert. Gängige Einheit von μ ist Reparaturen pro Stunde. Der Zusammenhang zwischen MTTR und μ ist wie folgt gegeben:

$$MTTR = \frac{1}{\mu}$$

MTBF

MTBF (mean time between failure) gibt die durchschnittliche Zeit zwischen zwei Fehlern innerhalb eines Systems an. Dies ist konzeptuell ein Unterschied zu MTTF, da bei MTTF die durchschnittliche Zeit bis zum ersten Fehler angegeben wird. Experimentell kann MTBF durch das parallele Ausführen von N identischen Systemen über einen Zeitraum T hinweg bestimmt werden. Zunächst muss nun die Anzahl der durchschnittlichen Fehler n_{avg} bestimmt werden. Dazu muss die Anzahl der aufgetretenen Fehler n_i für jedes System i berechnet werden. Daraus und aus der Kenntnis von T kann MTBF berechnet werden:

$$n_{avg} = \sum_{i=1}^{N} \frac{n_i}{N}$$

$$MTBF = \frac{T}{n_avg}$$

Falls angenommen werden kann, dass alle Reparaturen das System wieder in den Anfangszustand versetzen, kann der Zusammenhang zwischen MTBF und MTTF einfach bestimmt werden. Sobald das System repariert wurde, wird es durchschnittlich die Zeitdauer MTTF korrekt arbeiten. Im Anschluss daran muss das System entsprechend der Zeitdauer MTTR repariert werden. Somit kann MTBF durch folgende Gleichung angegeben werden:

$$MTBF = MTTF + MTTR$$

Berechnungsmodelle

Problem: für typische Systeme sind die Fehlerraten (z.B. $\lambda = 10^{-9} errors/hour$) so klein, dass eine experimentelle Bestimmung nicht möglich ist. Allerdings sind die Zuverlässigkeiten bzw. Fehlerraten der Komponenten bekannt.

Lösung: Modelle zur Berechnung der Zuverlässigkeit

- Kombinatorische Modelle
- Markow-Modelle

Kombinatorische Modelle

Dieses Modell basiert auf der Berechnung von Wahrscheinlichkeiten, dass ein System operabel ist, basierend auf den bekannten Fehlerraten von Subkomponenten.
Es existieren zwei Systemmodelle (Serienschaltung, Parallelschaltung) die benutzt werden können, um das System zu modellieren.

Bei der Serienschaltung wird angenommen, dass für eine korrekte Ausführung alle Komponenten korrekt arbeiten müssen. Im Gegensatz dazu muss bei der Parallelschaltung zur korrekten Funktion nur eine Komponente korrekt arbeiten.

Serienschaltung

Sei $C_{iw}()$ das Ereignis, dass die Komponenten C_i zum Zeitpunkt t korrekt arbeitet, $R_i(t)$ die Zuverlässigkeit der Komponente C_i und $R_{series}(t)$ die Zuverlässigkeit der Serienschaltung. Weiterhin wird angenommen, dass die Serie aus N Komponenten besteht. Die Zuverlässigkeit der Serienschaltung kann mathematisch wie folgt durch die Wahrscheinlichkeit P ausgedrückt werden:

$$R_{series}(t) = P\{C_{1w}(t) \cap C_{2w}(t) \cap ... \cap C_{Nw}(t)\}$$

Unter der Annahme, dass alle Ereignisse $C_{iw}(t)$ unabhängig voneinander sind, kann die Zuverlässigkeit durch

$$R_{series}(t) = R_1(t)R_2(t)...R_N(t) \text{ oder } R_{series} = \prod_{i=1}^{N} R_i(t)$$

berechnet werden.

Parallelschaltung

Ähnlich zu der Herleitung der Zuverlässigkeit der Serienschaltung kann auch die Zuverlässigkeit der Parallelschaltung berechnet werden. Sei $C_{if}()$ das Ereignis, dass die Komponenten C_i zum Zeitpunkt t fehlerhaft ist und $Q_i(t)$ die Unzuverlässigkeit der Komponente C_i $(R_i(t) = 1 - Q_i(t))$ und $R_{parallel}(t)$ die Zuverlässigkeit der Parallelschaltung. Die Zuverlässigkeit der Parallelschaltung kann mathematisch wie folgt durch die Wahrscheinlichkeit P ausgedrückt werden:

$$R_{parallel}(t) = 1 - P\{C_{1f}(t) \cup C_{2f}(t) \cup ... \cup C_{Nf}(t)\}$$

Unter der Annahme, dass alle Ereignisse $C_{if}(t)$ unabhängig voneinander sind, kann die Zuverlässigkeit durch

$$R_{parallel}(t) = 1 - Q_1(t)Q_2(t)...Q_N(t)$$
oder
$$R_{parallel} = 1 - \prod_{i=1}^{N} Q_i(t) = 1 - \prod_{i=1}^{N}(1 - R_i(t))$$

berechnet werden.

m-von-n Systeme sind eine Spezialform von Parallelschaltungen und können ähnlich behandelt werden.

Nachteil kombinatorischer Modelle

- Annahme der statistische Unabhängigkeit
- Komplexe Systeme können oft nicht modelliert werden
- Fehlererkennung kann schwer in kombinatorisches Modell integriert werden
- Reparatur kann schwer in kombinatorisches Modell integriert werden

$\longrightarrow$Einführung von Markow-Modellen

Markow-Modelle

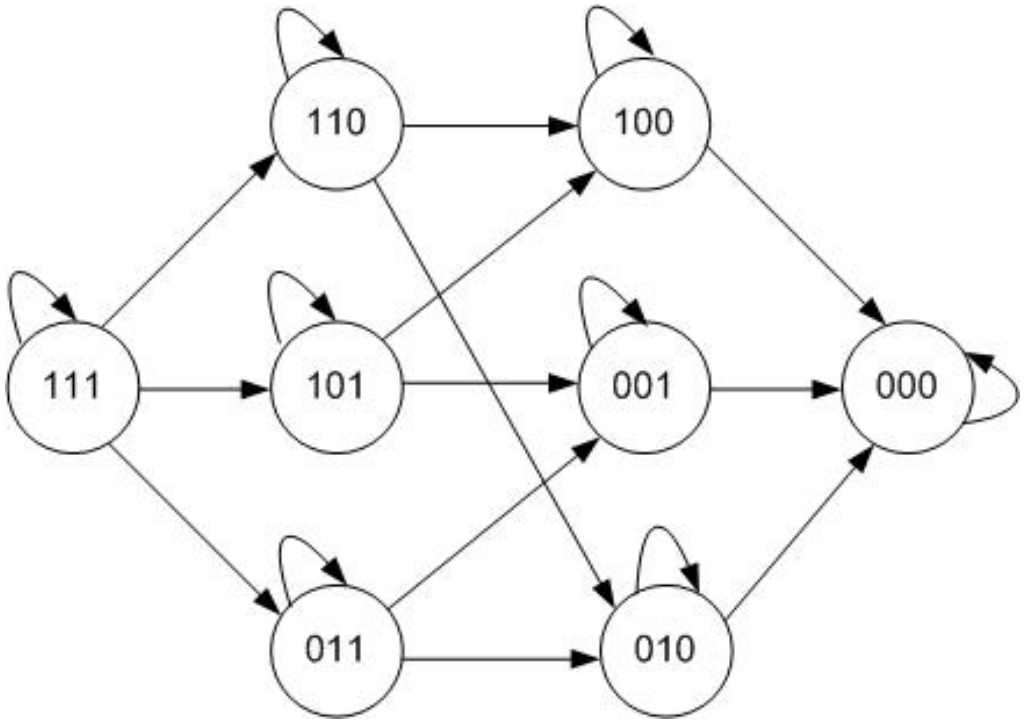
Hauptkonzepte:

- Zustände (states) $x \in X$: Zustände, die das System annehmen kann
- Zustandsübergänge (transition) $tr(x, x')$: Übergänge zwischen zwei Zuständen x und x' verknüpft mit einer Wahrscheinlichkeit $P(tr(x, x'))$(abhängig von der Wahrscheinlichkeit der Fehler, der Fehlerentdeckung und der Reparatur)

Markow-Annahme:
Wahrscheinlichkeit, dass sich das System zum Zeitpunkt $t + \Delta t$ im Zustand x befindet, ergibt sich aus der Summe der Produkte der Wahrscheinlichkeit der Vorgängerzustände zum Zeitpunkt tr und der Transitionswahrscheinlichkeit.

$$P(x, t + \Delta t) = \sum_{x' \in X} P(x', t) * P(tr(x', x))$$

Markow-Modell eines 2-von-3 Systems



2-von-3-System ohne Reparaturmöglichkeit

Übergangswahrscheinlichkeiten

Unter der Annahme, dass die Komponenten eine konstante Fehlerrate λ besitzen und unabhängig voneinander sind, ist die Wahrscheinlichkeit $P_f(\Delta t)$, dass eine Einheit zur Zeit $t + \Delta t$ ausgefallen ist, falls sie zum Zeitpunkt t korrekt arbeitete durch

$$P_f(\Delta t) = 1 - e^{-\lambda \Delta t}$$

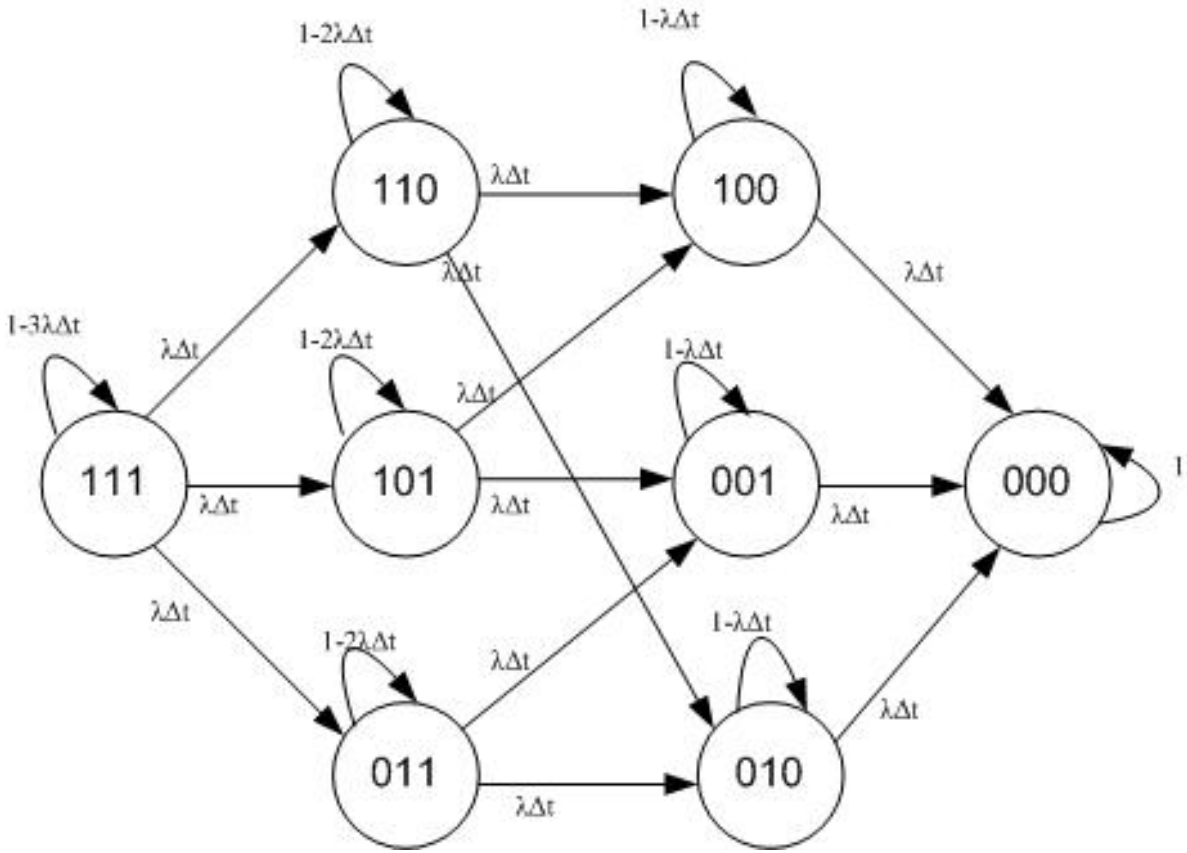
gegeben.
Durch Entwicklung erhalten wir:

$$P_f(\Delta t) = 1 - e^{-\lambda \Delta t} = 1 - \left[1 + (-\lambda \Delta t) + \frac{(-\lambda \Delta t)^2}{2!} + ...\right] = \lambda \Delta t - \frac{(-\lambda \Delta t)^2}{2!} - ...$$

Wird Δt so gewählt, dass $\lambda \Delta t << 1$ reduziert sich der Ausdruck zu

$$P_f(\Delta t) = \lambda \Delta t$$

Markow-Modell mit Wahrscheinlichkeiten

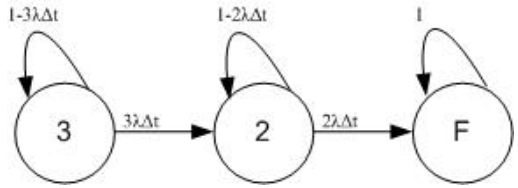


Reduziertes Markow-Modell

Oftmals können Zustände im Markow-Modell in Kategorien eingeordnet werden.
Im Beispiel wären dies

- der perfekte Zustand (111)
- der 1-Fehler-Zustand (110),(101),(011) und
- der Fehlzustand (100)(010)(001)(000)

Man spricht von einer Kollabierung der Zustände und einem reduzierten Markow-Modell.



Vorlesung Echtzeitsysteme - Fehlertoleranz und Zerberus

Christian Buckl

Lehrstuhl Embedded Systems and Robotics

Sommersemester 2005

Ziel dieser Stunde

Welche Fehlerquellen gibt es und wie wirken diese sich aus?

Wann werden welche Fehlertoleranzmechanismen angewandt?

Welche Probleme treten bei der Implementierung auf?

Welche Mechanismen sollten im Zerberus System verwendet werden?

Überblick

- Fehlerquellen und ihre Auswirkungen
- Ziele des Zerberus Systems
- Fehlertoleranzmechanismen und ihre Verwendung

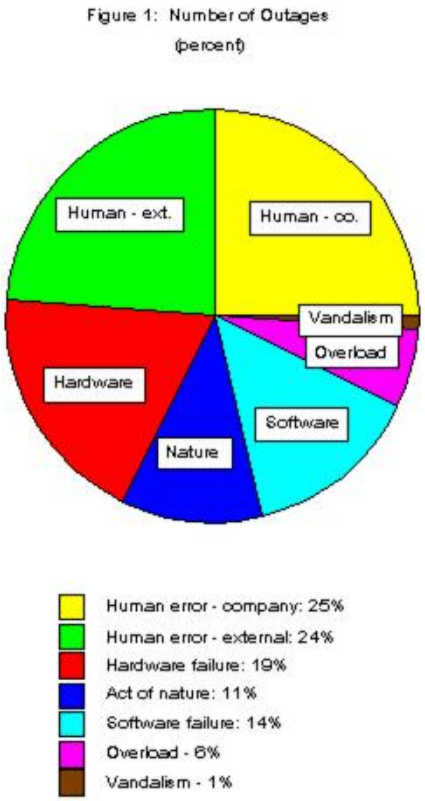
Fehlerquellen im öffentlichen Telefonnetz

Welche Ursachen können Fehler haben:

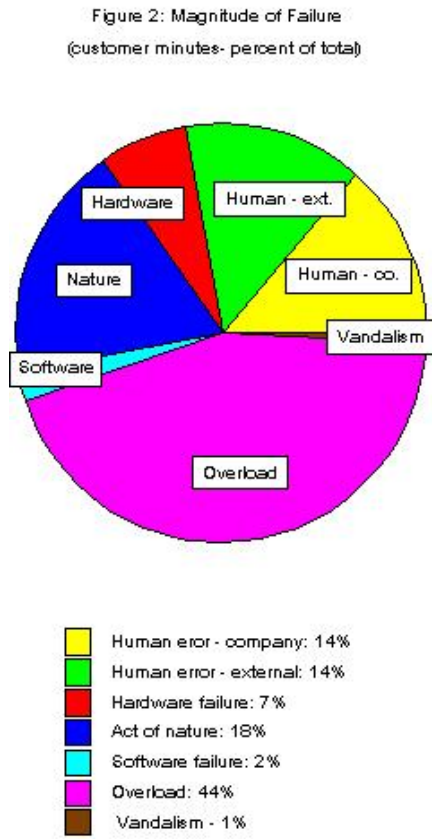
- Fehler durch Menschen (intern/extern)
- Hardwarefehler
- Softwarefehler
- Fehler verursacht durch die Natur
- Überlast
- Vandalismus

Weitere Informationen unter
http://hissa.ncsl.nist.gov/kuhn/pstn.html.

Auswirkungen von Fehlern



Anteil an Ausfällen



Schwere der Ausfälle
(in Benutzerminuten)

Hintergrund

Für die Entwicklung von sicheren und zuverlässigen Computersystemen existieren keine geeignete Modelle.

Dies führt

- zur ständigen Neuentwicklung von Fehlertoleranzmechanismen,
- zu fehlerbehafteten Systemen,
- erhöhten Entwicklungskosten,
- zum Verzicht auf den Einsatz von Computersystemen trotz eindeutiger Vorteile.

Ziele des Zerberus Systems

- Trennung der Implementierung von Funktionalität und Fehlertoleranzmechanismen
- Angebot an wiederverwendbaren Fehlertoleranzmechanismen
- Echtzeitfähigkeit
- Weitgehende plattformunabhängige Implementierung
- Verkürzung der Entwicklungszeiten
- Reduzierung der Entwicklungskosten
- Unterstützung bei der Zertifizierung
- Erweiterbarkeit

Welche Fehlertoleranzverfahren können für das Zerberus System benutzt werden?

Welche Fehlertoleranzmechanismen gibt es?

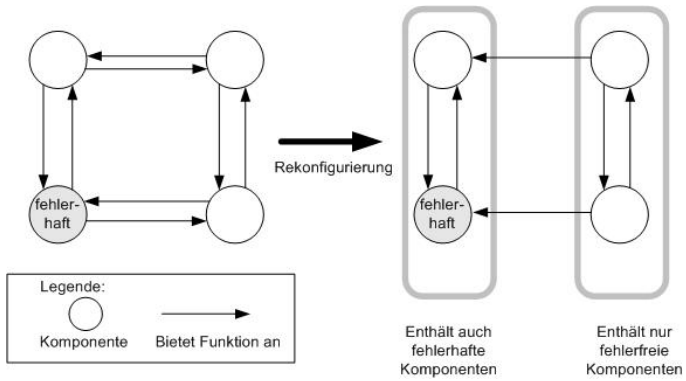
Welche Mechanismen kann/sollte man in einem System wie Zerberus verwenden?

Rekonfigurierung

Wiederholung: Rekonfigurierung unterteilt die Menge der Komponenten bei Auftreten eines Fehlers in eine Menge von fehlerfreien Komponenten und eine Menge eventuell fehlerbehafteter Komponenten.

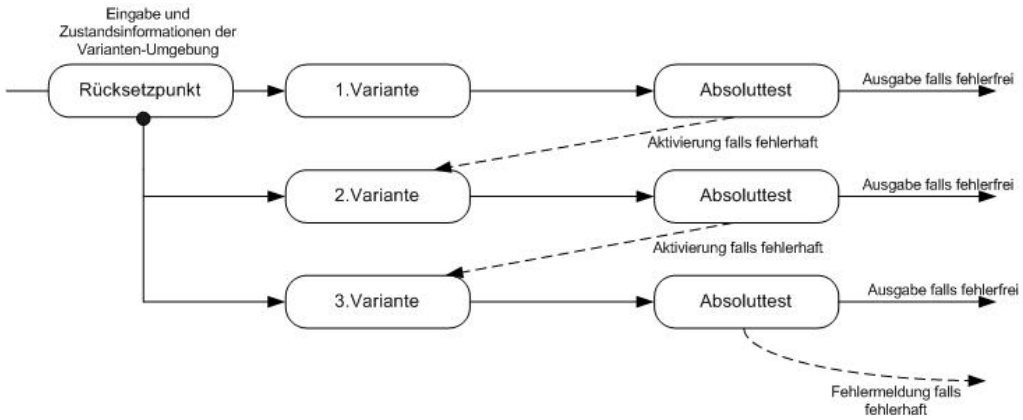
Wieso reicht Rekonfigurierung nicht für Fehlerbehandlung aus?

Bewertung: Rekonfigurierung kann höchstens zusätzlich im Zerberus System verwendet werden.



Rückwärtsbehebung

Wiederholung: Rückwärtsbehebung versetzt ein System in Zustand, den sie bereits in der Vergangenheit angenommen haben oder in einem konsistenten Zustand hätten annehmen können.



Implementierungsfragen

Allgemein:

- Wie oft/Wann werden Rücksetzpunkte angelegt?
- Nach welchem Schema sollen Rücksetzpunkte angelegt werden?
- Was/Wo wird gespeichert?
- Welche Fehler können erkannt werden?
- Wie können die Tests gestaltet werden?

Zerberus spezifisch:

- Können Rücksetzpunkte sinnvoll ohne Anwendungswissen erstellt werden?
- Welche Kenntnisse sind für eine gezielte Rücksetzlinienerstellung nötig?
- Können Absoluttests ohne Anwendungswissen erstellt werden?

Bewertung

\+ Transparent zu implementieren

\+ Sparsame Betriebsmittelverwendung (keine Redundanz nötig)

o Für sinnvolle Implementierung der Rücksetzpunkterstellung ist Anwendungswissen erforderlich → Entwickler muss mithelfen

\- Nicht echtzeitfähig wegen Wiederholungen

\- Schon während des Nutzbetriebs teure Rücksetzpunktanlegung

\- Nur Absoluttest verwendbar

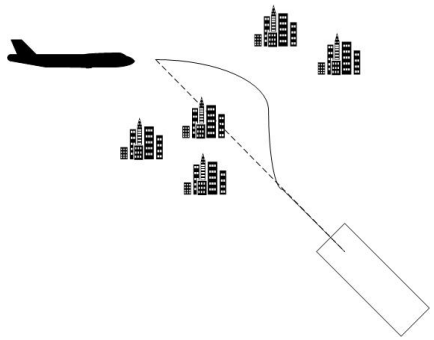
\- Tests müssen durch den Entwickler implementiert werden

→ Grundsätzlich geeignet aber nicht ausreichend (wegen mangelnder Echtzeitfähigkeit).

Vorwärtsbehebung

Wiederholung: Mechanismen, die keine Zustandsinformationen der Vergangenheit verwenden.

Basis der Verfahren sind Fehlfunktions-Annahmen und anwendungsspezifisches Wissen.



Bewertung

+ Minimaler zusätzlicher Aufwand im Ausnahme- und im Normalbetrieb

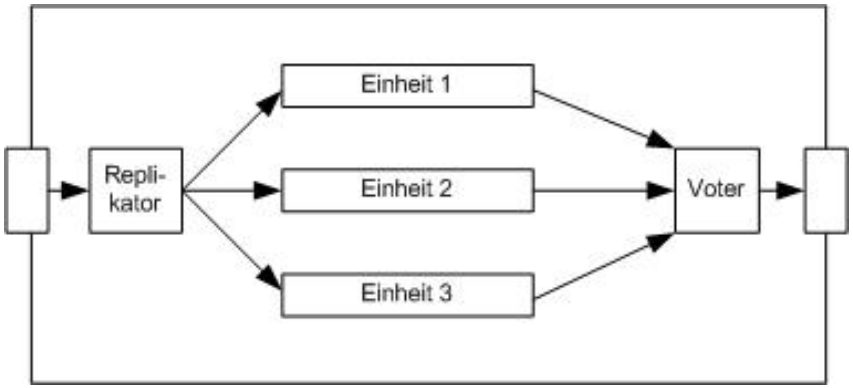
- Nur anwendungsspezifisch implementierbar
- Nur Absoluttests sind unterstützbar
- Oftmals nur degradierter Betrieb möglich

Bewertung: nicht geeignet.

Fehlermaskierung

Wiederholung: Fehlermaskierung bezeichnet Verfahren, die aus redundanten Ergebnissen die korrekten Ergebnisse auswählen.

Stichwörter: Replizierung, Votierung, Synchronisierung, Ausgrenzung, Integration, Ausgabe, Determinismus



Praktische Fragen zur Votierung

Welche Art der Entscheidung ist sinnvoll?

- Mehrheitsentscheidung
- Meiststimmenentscheidung
- Paarentscheidung
- Einstimmigkeitsentscheidung

Wie wird das Voting realisiert?

- Zentral vs. verteilt
- Realisiert in Hardware vs. in Software

Praktische Fragen zur Votierung / Synchronisation

Sind die Ergebnisse der redundanten Prozesse deterministisch?
Quellen des Indeterminismus:

- Prozesskontext
- unterschiedl. Hardware
- unterschiedliche Implementierung (N-Versions-Programmierung)
- Synchronisationsprobleme

Wie kann die Synchronisation realisiert werden?

- Zentrale Steuerung
- Steuerung durch den Maskierer
- Steuerung durch Auftragsrate
- verteilt

Praktische Fragen zum Redundanzgrad

Faktoren:

- MTTR/MTBF
- MTTF

N-Versionsprogrammierung hilft nur bei kleinem N.

Praktische Fragen zur Reparatur

Wie kann das System repariert werden?
→ Ausgliederung und zeitunkritische Durchführung der Reparatur, die Reparatur ist dabei zumeist anwendungsabhängig

Wie kann eine anwendungsunabhängige Integration in das System realisiert werden (state synchronisation)?
→ Trennung von Funktionalität und inneren Zustand.

Problem: Eingabe und Ausgabe

Wie können die Eingaben repliziert werden?

- Replizierung durch Hardware
- Replizierung durch Softwaremodul (single point of failure)

Wie kann eine korrekte Ausgabe sichergestellt werden?

- Ausgabe liegt im Perfektionskern.
- Redundante Ausgabe.
- Ausgabe mit Rückwärtsbehebung.

Fragen bzgl. Zerberus

- Wie kann die Synchronisierung sichergestellt werden?
- Wie kann mit nichtdeterministischen Prozessen umgegangen werden?
- Können Algorithmen unabhängig vom Redundanzgrad entworfen werden?

Bewertung

+ Redundanzgrad kann frei gewählt werden
+ Maskierung echtzeitfähig
+ Maskierung ist applikationsunabhängig implementierbar
+ Insgesamt relativ einfach zu implementieren
+ Relativtests sind anwendbar
+ höchste Fehlertolerierungsrate aller Fehlertoleranzmechanismen

o Bei indeterministischen Prozessen sind Informationen durch den Entwickler nötig.
o Für eine Reparatur sind zumeist Informationen durch den Entwickler nötig.

Fehlerkorrektur

Wiederholung: Durch Informationsredundanz können verfälschte Ergebnisse wieder korrigiert werden.

Beispiele: Hamming-Code

Anwendungsbereiche: Übertragung und Speicherung von Daten

Bewertung

+ anwendungsunabhängig implementierbar

- Einsatzbereiche sehr beschränkt
- hoher Mehraufwand nötig

Zusammenfassung

Fehlermaskierung ist das erfolgsversprechendste Konzept.
Es können prinzipiell alle Fehlerarten erkannt und toleriert werden:

- Hardwarefehler
- Softwarefehler
- Entwurfsfehler
- Laufzeitfehler

Aber auch die anderen Fehlertoleranzmechanismen sollten unterstützt werden, um deren Vorteile (z.B. sparsamerer Betriebsmittelbedarf) auszunutzen.

Vorlesung Echtzeitsysteme - Zerberus System

Prof. Alois Knoll

Lehrstuhl Embedded Systems and Robotics

Sommersemester 2005

Inhalt

- Motivation
- Entwicklungsmodell
- Sprache Zerberus
- Laufzeitsysteme
- Ausblick

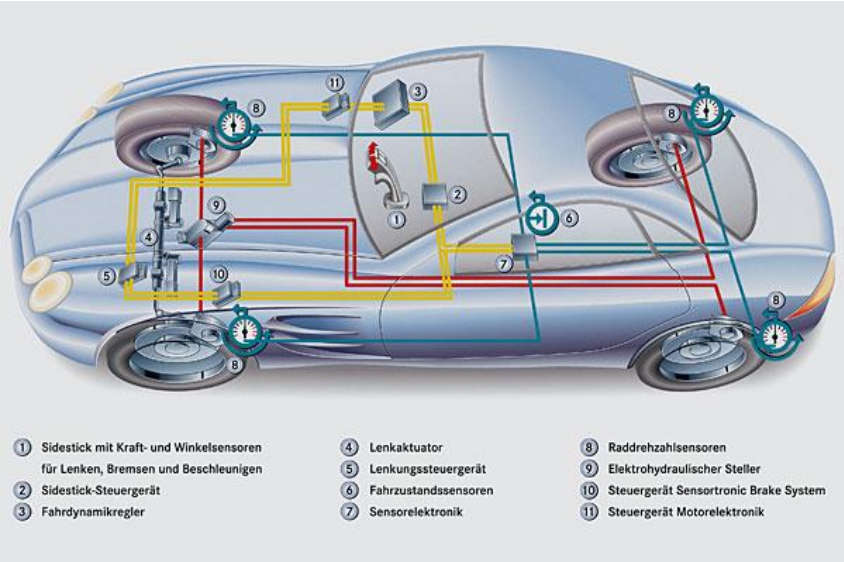
Literatur

- T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Embedded control systems development with giotto. Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), S.64-72, 2001.
- T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. Proceedings of the First International Workshop on Embedded Software (EMSOFT), S.166-184, 2001.
- H. Kopetz and G. Bauer. The Time-Triggered Architecture. Proceedings of the IEEE, S.112-126, Jan. 2003
- L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. J. ACM, S.52-78, 1985
- S. Poledna, A. Burns, A. Wellings, and P. Barrett. Replica determinism and flexible scheduling in hard real-time dependable systems. IEEE Transactions on Computers, S.100-110, Feb. 2000
- Christian Buckl, Zerberus Language Specification Version 1.0, Technischer Bericht 2005

Einsatzzahlen von Computersystemen in sicherheitskritischen Bereichen steigen

Beispiele:

Drive-By-Wire



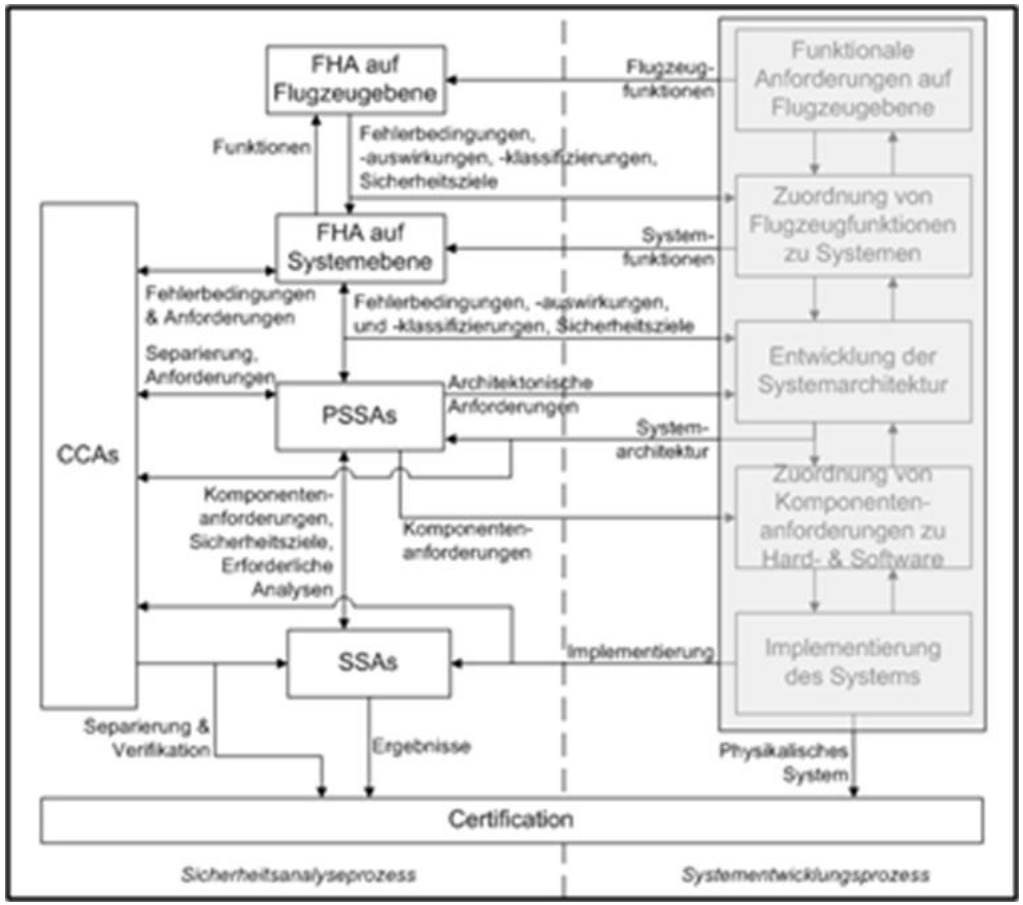
Computerassistenz in der Medizin
z.B. CaVaCo

Gegenwärtige Entwicklungsstandards

- FT-Mechanismen werden immer wieder neu implementiert bzw. neu erfunden.
- Entwicklung richtet sich zumeist nur nach firmeninternen Standards
- Mischung der Implementierung von FT-Mechanismen und Anwendungsfunktionalität

$\longrightarrow$ Es wird ein Entwicklungsmodell für sichere und zuverlässige Echtzeitsysteme benötigt.

Typischer Zertifizierungsprozess (ARP 4754)



ARP: Aerospace Recommended Practices, FHA: Failure Hazard Assessment, PSSA: Preliminary System Safety Assessment, SSA: System Safety Assessment, CCA: Common Cause Analysis

Minimalunterlagen für Zertifizierung

- Zertifizierungsplan
- Konfigurationsindex
- Zertifizierungszusammenfassung

Zertifizierungsplan

Der Zertifizierungsplan enthält neben einer funktionalen und operationellen Beschreibung des Systems und einer Zusammenfassung aller durchgeführten Analysen auch eine Übersicht über andere Zertifizierungspläne, die möglicherweise von Bedeutung für den vorliegenden sind. Außerdem werden diejenigen Elemente des Entwurfs beschrieben, mit denen Sicherheitsziele abgedeckt werden sollen sowie neue Technologien, die implementiert werden sollen. Ein weiterer Bestandteil des Zertifizierungsplans ist eine Liste aller Unterlagen, die für die Zertifizierung eingereicht werden sollen, ergänzt durch einen Zeitplan für die Zertifizierung sowie eine Aufführung aller verantwortlichen Personen oder Einrichtungen, die mit zertifizierungsrelevanten Aktivitäten befasst sind.

Konfigurationsindex

Im Konfigurationsindex wird die Konfiguration jeder Systemkomponente identifiziert und aufgeführt. Insbesondere wird auch die zugehörige Software definiert sowie Querverbindungen zu anderen Komponenten und Schnittstellen mit anderen Systemen. Schließlich werden hier auch sicherheitsrelevante Verfahrensweisen oder Beschränkungen für den Betrieb oder die Wartung der Komponenten beschrieben.

Zertifizierungszusammenfassung

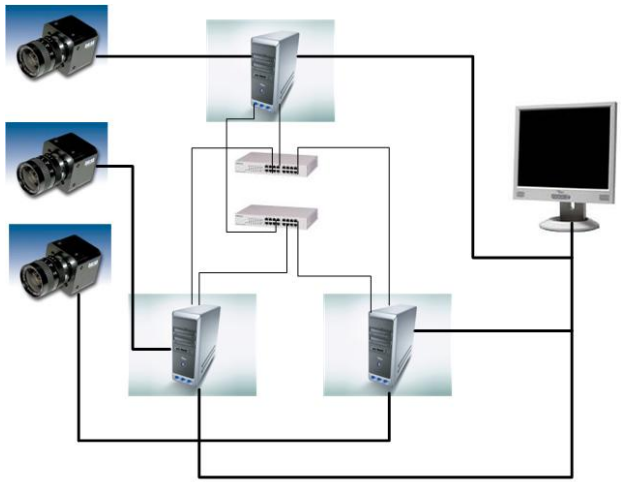
Die Zertifizierungszusammenfassung enthält neben der Feststellung, dass das zu zertifizierende System die Anforderungen zur sicherheitstechnischen Anforderungen einhält auch eine Beschreibung, wie die Einhaltung dieser Anforderungen überprüft wurde sowie eine Beschreibung von möglicherweise noch offenen Problemen, die Einfluss auf die Funktionalität oder die Sicherheit des Systems haben können.

Ziele des Zerberus Systems

- Kostenreduzierung
- Trennung von Funktionalität und Fehlertoleranzmechanismen
- Wiederverwendung von Fehlertoleranzmechanismen
- Erfüllung harter Echtzeitanforderungen
- Erleichterung der Zertifizierung

Implementierungsansatz

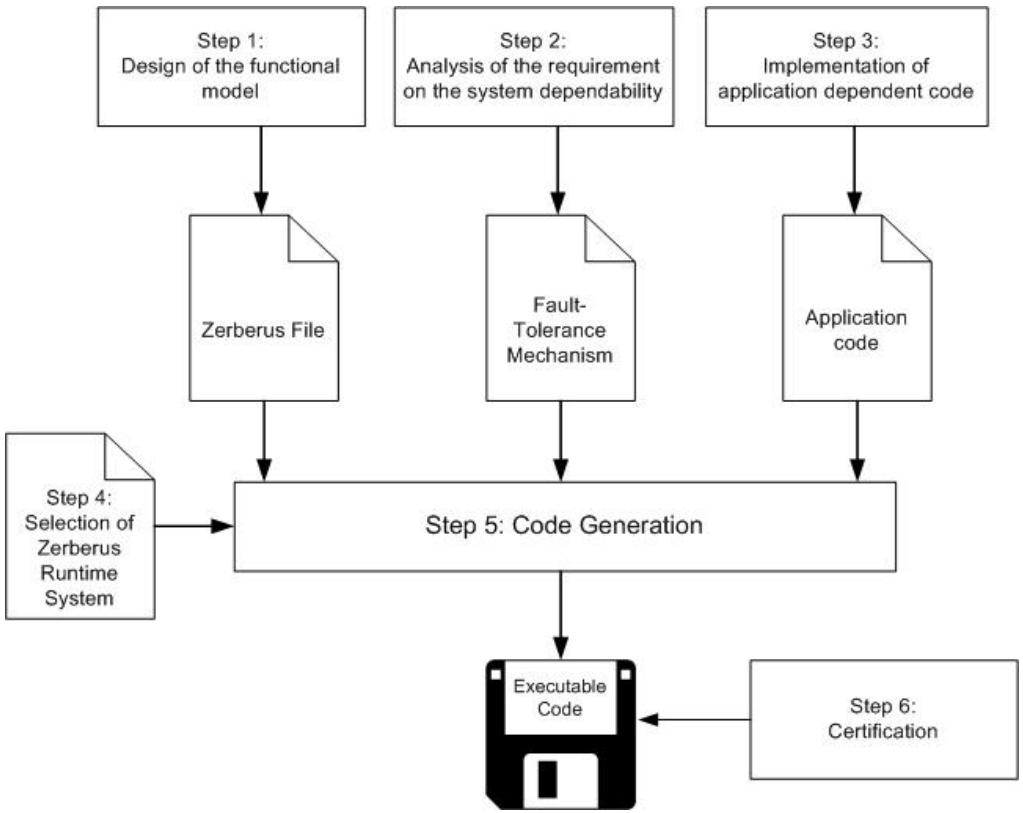
- TMR-System aus Standardhardwarekomponenten
- Alle Einheiten sind mit der Umgebung verbunden
- Unterstützung unterschiedlichster Sensorik und Aktorik
- Hohe Anforderungen an erreichbare Regelungszeiten
- Generalität (keine Beschränkung auf spezielle Hardware, Betriebssysteme oder Programmiersprachen)
- Erweiterbarkeit



Was ist das Zerberus System

- Entwicklungsmodell
- Werkzeuge zur Unterstützung der einzelnen Schritte
- Sprache zur plattformunabhängigen Spezifikation des formalen Modells der Anwendung
- Laufzeitsysteme
- Protokolle für Fehlertoleranzmechanismen (Votierung, Synchronisation, Integration)

Entwicklungsprozess



Schritt 1: Spezifikation des formalen Modells

Ziel: Plattformunabhängige Spezifikation des formalen Modells der Anwendung in der Sprache Zerberus:

- Identifikation der funktionalen Einheiten
- Bestimmung der Abhängigkeiten dieser Einheiten
- Spezifikation der zeitlichen Bedingungen

Ergebnis: Formales Modell beschrieben durch Zerberus Code (zerberus file).

Anforderungen aufgrund der Fehlertoleranzmechanismen

Das formale Modell dient als Basis zur automatischen Realisierung der Fehlertoleranzmechanismen. Aus diesem Grund müssen folgende Anforderungen durch die Sprache Zerberus erfüllt werden.

- Deterministische Ausführung der redundanten Einheiten: Nur bei einer deterministischen Ausführung der redundanten Einheiten ist eine Votierung möglich. Allerdings schränkt eine strikte deterministische Ausführung die Entwurfsmöglichkeiten bei der N-Versionsprogrammierung stark ein. $\rightarrow$ Es muss Zeitpunkte geben, zu denen eine korrekte Votierung möglich ist.
- Deterministische Zeitpunkte: Um Synchronisations- und Votierungsalgorithmen verteilt (ohne Master) auf den Rechnern implementieren zu können, müssen diese Zeitpunkte vorab bekannt sein.
- Trennung des Zustandes von der Funktionalität: Um eine automatische Realisierung der Fehlertoleranzmechanismen zu erreichen, muss der Zustand eines Systems extrahiert werden können. Dies wird am besten durch eine Trennung des Zustands von der Funktionalität im Modell erreicht. Auf der Basis der Zustandswerte der einzelnen Rechner kann dann eine Votierung durchgeführt werden.

Grundlagen der Sprache Zerberus

Die Sprache Zerberus

- basiert auf Giotto (Berkeley) einer formalen Sprache zur Spezifikation des formalen Modells verteilter Systeme,
- ist zeitgesteuert und
- erlaubt die plattformunabhängige Spezifikation des formalen Modells.

Giotto

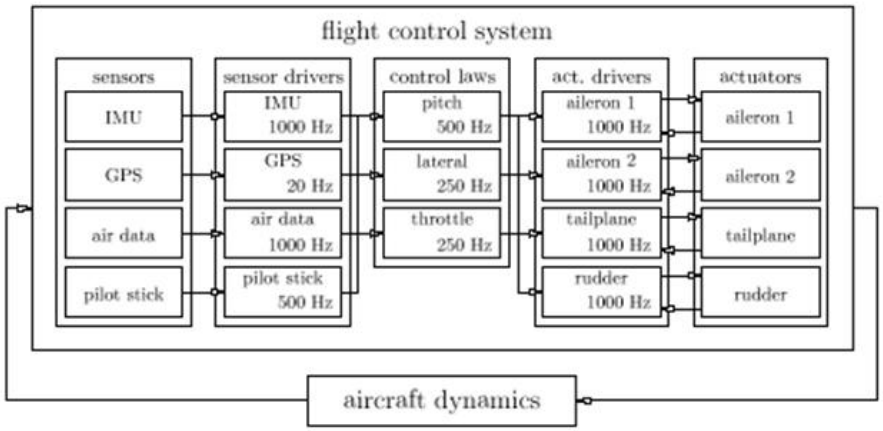
Zielsetzung:

- Programmiermodell für verteilte Echtzeitsysteme
- Anwender muss nur noch den eigentlichen Anwendungscode implementieren
- Die Zuweisung der Prozesse der einzelnen Rechner erfolgt automatisch durch den Compiler

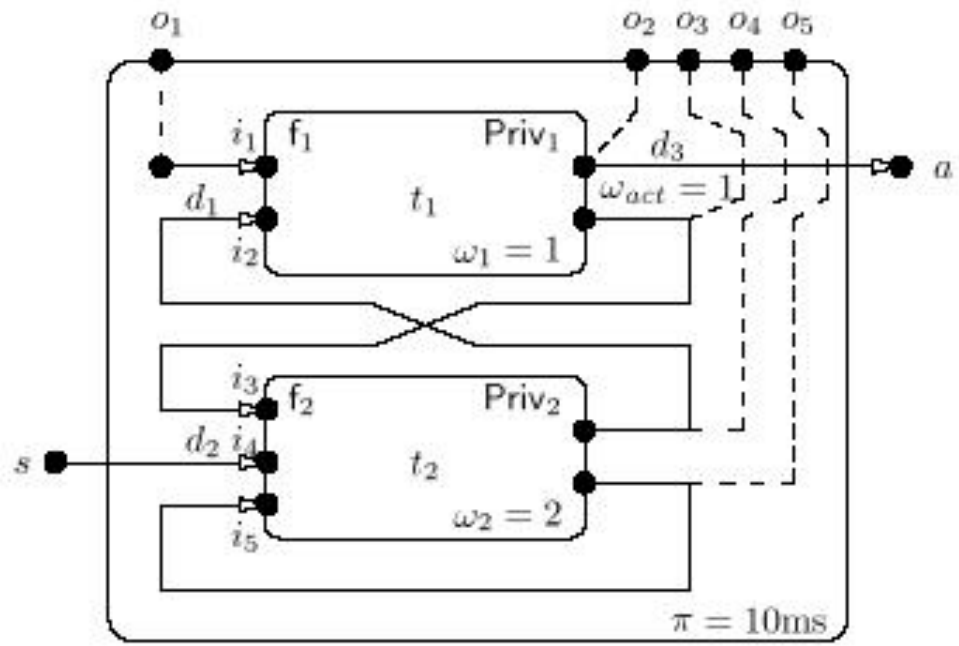
Ergebnisse:

- Sprache Giotto zur Beschreibung des formalen Modells
- Giotto basiert auf Fixed Logical Execution Times Konzept (die Ausführung von Berechnungen dauert immer eine fest vorgegebene Zeit)
- Codegenerator, der E-Code erzeugt
- Embedded Machine: Virtuelle Maschine zur Ausführung von Code auf verschiedenen Plattformen

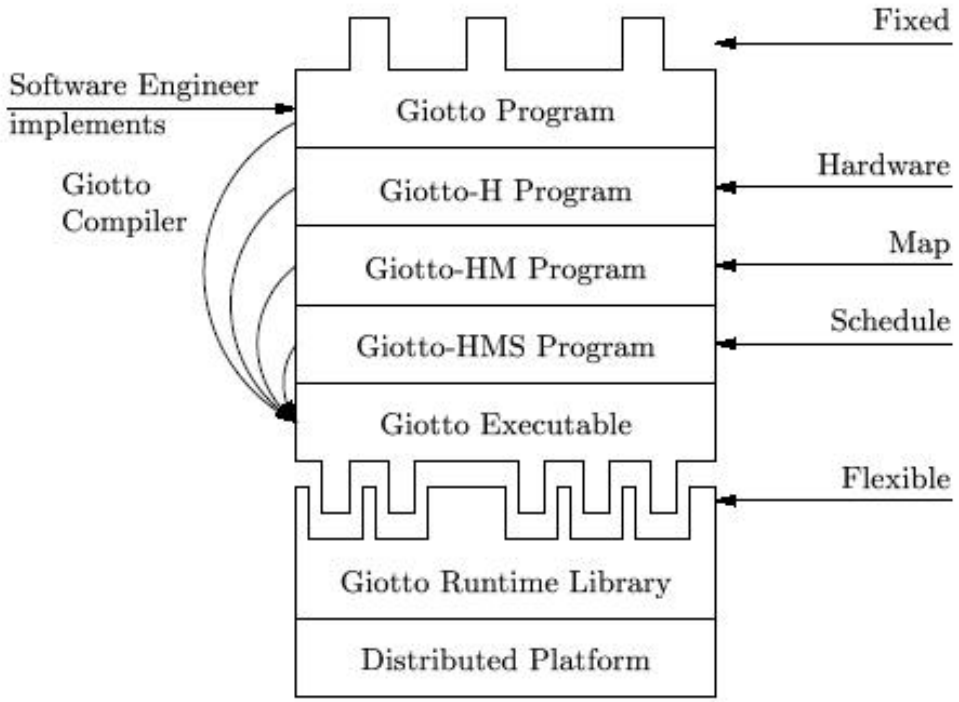
Giotto-Beispiel: Helikoptersteuerung



Giotto: Ein Giotto-Mode



Annotated Giotto



Schritt 2: Wahl der Fehlertoleranzmechanismen

Ziel: Analyse des Systemanforderungen in Bezug auf Zuverlässigkeit und Sicherheit und Wahl der geeigneten Fehlertoleranzmechanismen.

- Strukturelle Hardwareredundanz (gefordert)
- Diversität in der Hardware
- Diversität in der Software (n-Versions-Programmierung)

Weitere Fehlertoleranzmechanismen sollen unterstützt werden.

Ergebnis: Gewünschte Fehlertoleranzmechanismen, in Zukunft Beschreibung der Fehlertoleranzmechanismen in eigener Sprache (siehe Ausblick)

Schritt 3: Implementierung des rein anwendungsabhängigen Codes

Ziel: Implementierung des rein anwendungsabhängigen Codes für die gewünschte(n) Plattform(en) (Hardware, Betriebssystem, Programmiersprache)

- Nur der rein anwendungsabhängige Code muss implementiert werden.
- Fehlertoleranzmechanismen, sowie die Umsetzung des formalen Modells werden durch die Laufzeitsysteme realisiert.

→ Da nur der anwendungsabhängige Code implementiert werden muss, ist der zusätzliche Aufwand für N-Versions-Programmierung stark reduziert.

Ergebnis: Code für die entsprechenden Plattformen

Schritt 4: Wahl des Laufzeitsystems

Ziel: Wahl der zu verwendenen Laufzeitsysteme. Laufzeitsysteme realisieren für eine bestimmte Plattform (Hardware, Betriebssystem, Programmiersprache):

- die Synchronisation,
- die Votierung,
- die Integration,
- und die Ausführung des formalen Modells.

Es werden für diverse Plattformen Laufzeitsysteme angeboten.
Aktuell sind folgende Kombinationen verfügbar:

- VxWorks 5.5 / C
- VxWorks 5.5 / C++ (in Arbeit)

Zudem sind SingleVersions (Systeme ohne Fehlertoleranz) und die Migration auf VxWorks 6.0, sowie auf andere Betriebssysteme in Arbeit bzw. angedacht.

Ergebnis: Zu verwendende Laufzeitsysteme.

Erweiterbarkeit durch Zerberus Tags

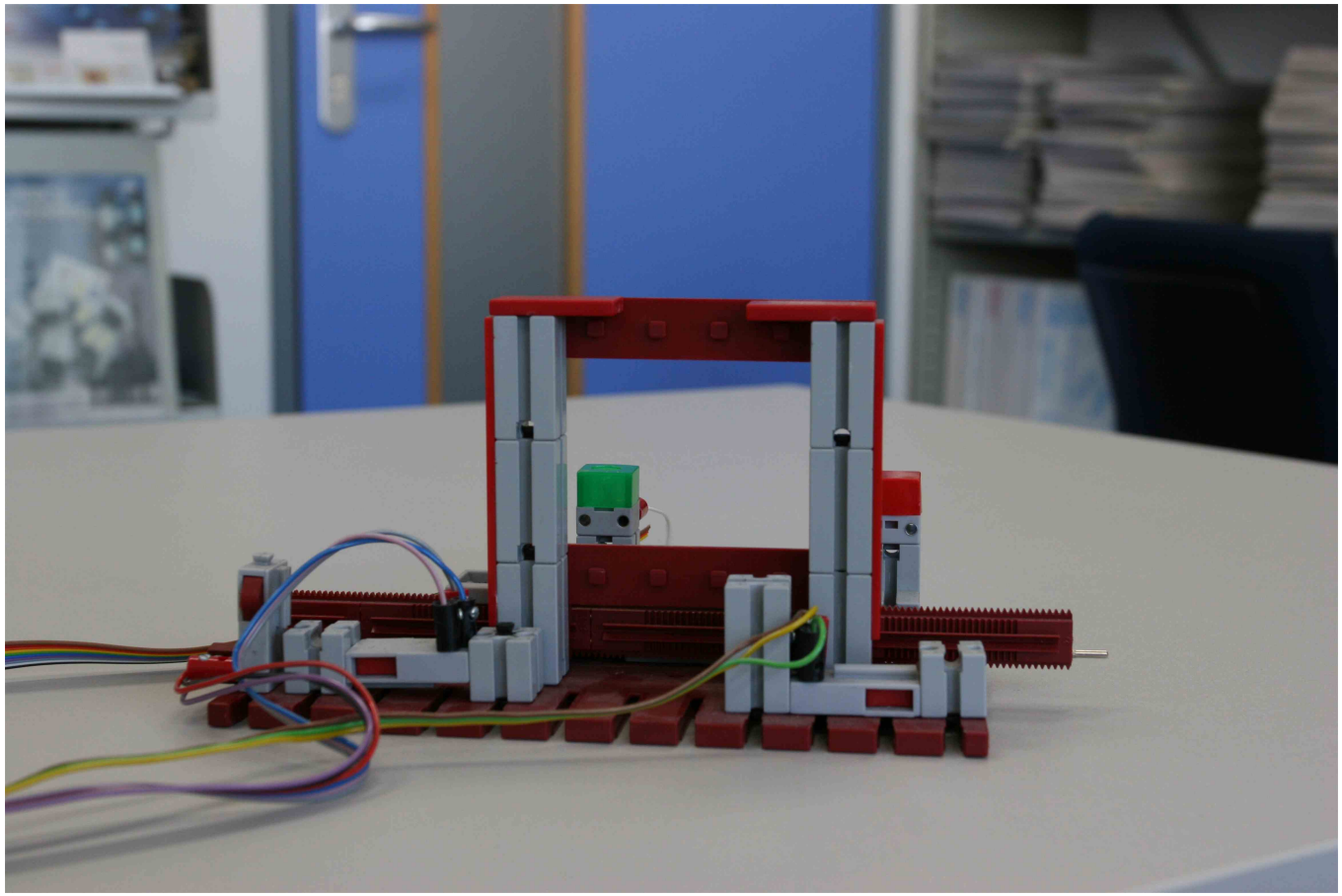
Wird für die gewünschte Plattform kein Laufzeitsystem angeboten, so kann dieses selbst implementiert werden.

- Die Protokolle für die Fehlertoleranzmechanismen werden angeboten.
- Zerberus Tags erlauben eine plattformunabhängige Implementierung des Laufzeitsystems.

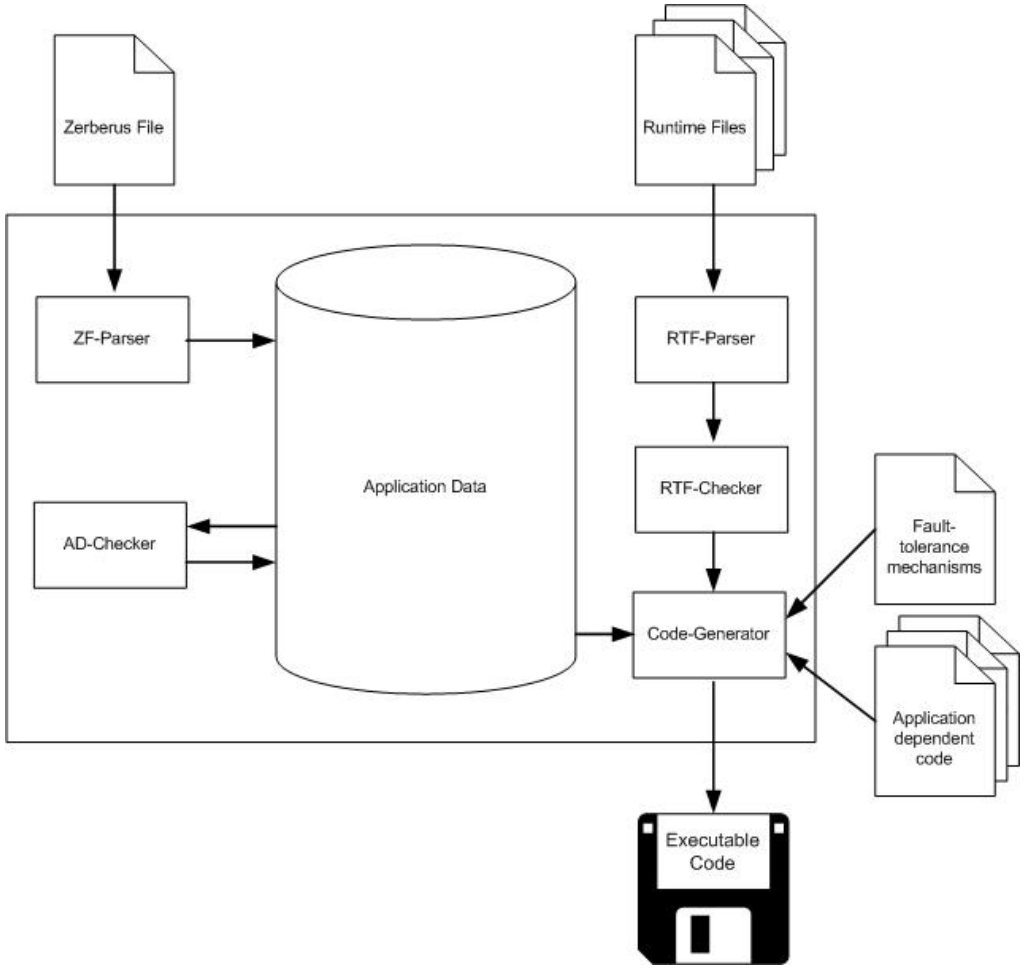
Zerberus Tags:

- Markieren Stellen im Code, die abhängig von der Anwendung ersetzt werden müssen.
- Codegenerator genereriert aus Laufzeitsystemcode mit Zerberus Tags ähnlich wie ein Präprozessor compilierbaren Code in der jeweiligen Programmiersprache.

Demonstration der Zerberus Tags



Schritt 5: Codegenerierung



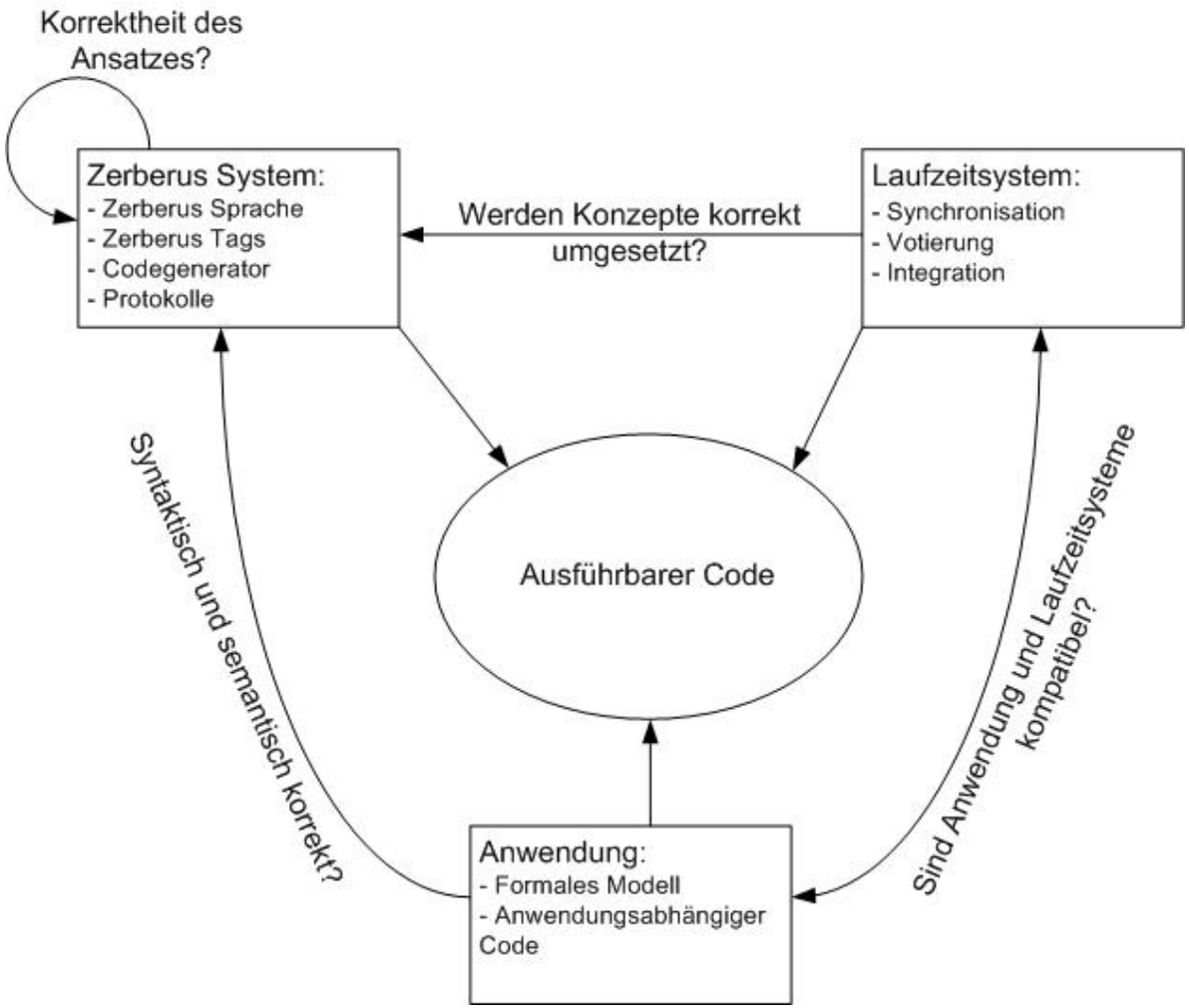
Schritt 6: Zertifizierung

Die Zertifizierung kann durch den dreiteiligen Aufbau unterstützt werden:

1. Zertifizierung des Zerberus Systems: Korrektheit des Zerberus System Ansatzes (Sprache, Fehlertoleranzmechanismen) und des Codegenerators.
2. Zertifizierung des Laufzeitsystems: Korrektheit der Umsetzung der Protokolle und der Ausführung des formalen Modells.
3. Zertifizierung der Anwendung: Korrektheit des vom Entwickler implementierten Codes, Kompatibilität der Anwendung mit dem verwendeten Laufzeitsystems (z.B. in Bezug auf die geforderten Regelungszeiten).

Ansatz: Werden ein zertifiziertes Zerberus System und zertifizierte Laufzeitsysteme benutzt, so beschränkt sich der Zertifizierungsprozess auf Schritt 3.

Zertifizierung

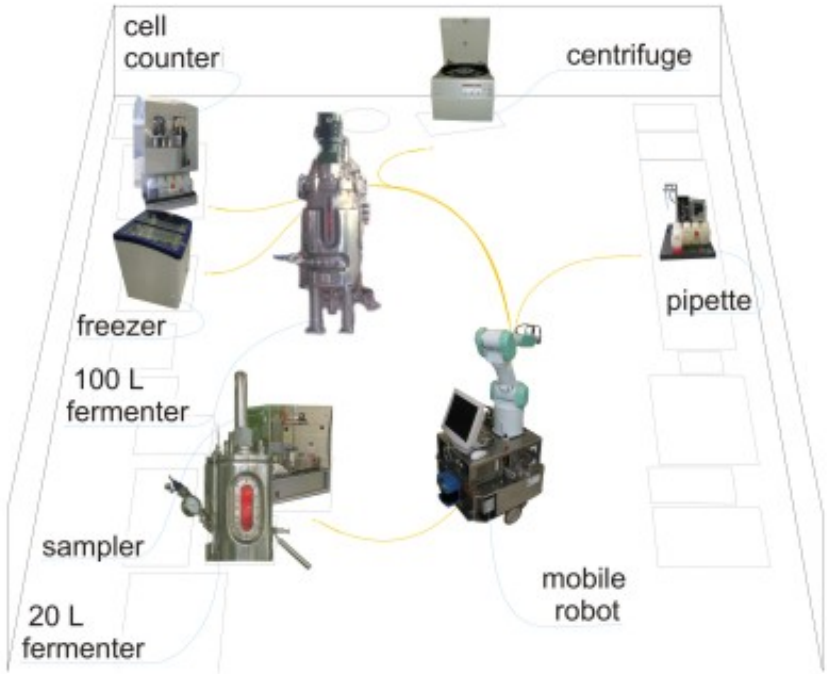


Sprache Zerberus

Objekte der Sprache Zerberus im Überblick

- Ports: Eindeutige Speicherplätze, Kommunikation in Zerberus erfolgt ausschließlich über Ports, der Zustand eines Systems lässt sich durch die Werte der Ports beschreiben.
- Tasks: Periodisch aufgerufenen Funktionen, Tasks stellen die eigentliche Funktionalität der Anwendung dar.
- Sensoren: Periodisch aufgerufene Funktionen zum Einlesen von Werten aus der Umgebung.
- Aktoren: Periodisch aufgerufene Funktionen zum Schreiben von Werten an die Umgebung.
- Modes: Sammlung von Tasks, Sensoren und Aktoren, die gleichzeitig auf dem Systen ausgeführt werden.
- Modechanges: Binäre Funktionen, die über einen Wechsel des aktuellen Modus entscheiden.
- Guards: Binäre Funktionen, die über das Starten eines assoziierten Tasks entscheiden.

Anwendungsbeispiel



Beispiel: Navigation in einem biotechnologischem Labor

- Roboter soll sich im Labor ohne Kollisionen bewegen.
- Verschiedene Arbeitsschritte müssen vollführt werden (z.B. Holen eines Reagenzglases aus der Zentrifuge und Platzierung der Probe im Gefrierschrank)
- Sensorik: WLAN (zur Übermittlung von Aufträgen), Funk (zur Bestimmung der Position), Kamera (zur Identifikation von Objekten)
- Aktorik: Ansteuerung der Motoren, sowie der Gelenkservos

Tasks:

- Periodisch aufgerufene sequentielle Funktionen.
- Tasks stellen die eigentliche Funktionalität der Applikation dar.
- Aufruffrequenz wird durch den Entwickler spezifiziert.
- Synchronisationspunkte zwischen Tasks sind nicht erlaubt.
- Die Ausführung der Tasks erfolgt transparent für den Entwickler: d.h. die Funktion verhält sich so, als wird sie zu Begin des Aufrufintervals gestartet und zum Ende des Intervals beendet.
- Der Entwickler muss garantieren, dass ein erfolgreiches Scheduling möglich ist: Worst Case Execution Time (WCET).

Tasks im Anwendungsbeispiel

Grundsätzlich können zwei Hauptfunktionen identifiziert werden:

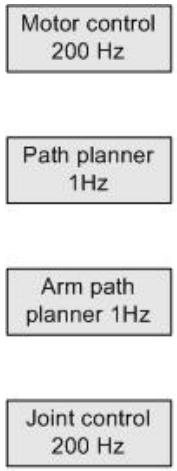
- Pfadplaner für den Roboter
- Pfadplaner für den Arm

Da beide Funktionen aufgrund der Komplexität der Berechnung unter Umständen sehr lange Berechnungszeiten haben, werden zur feineren Ansteuerung noch zwei weitere Tasks eingeführt:

- Motorensteuerung
- Gelenksteuerung

Diese Funktionen sollen Ergebnisse der Planer übernehmen und hochfrequent Ergebnisse an die Umwelt ausgeben.

Tasks im Anwendungsbeispiel

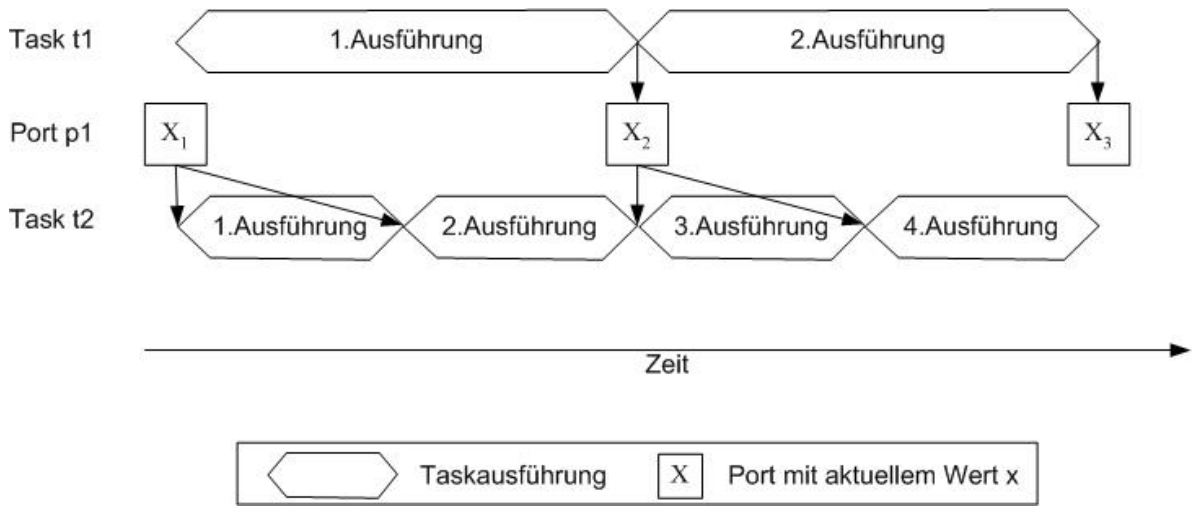


Ports

- Ports sind Plätze im Speicher zur Kommunikation zwischen Tasks.
- Jeder Port hat einen festgelegten Typ.
- Jeder Typ wird durch die Größe des Speicherplatzes und die Repräsentation der Werte festgelegt → plattformunabhängig.
- Werte dieser Speicherplätze stellen den Zustand des Systems dar → Ports bilden die Basis der Votierungsalgorithmen.
- Der Zugriff auf die Speicherplätze erfolgt zeitgesteuert (Lesezugriffe immer zu Beginn einer Taskperiode, Schreibzugriffe zum Ende der Taskperiode) und deterministisch (Schreibzugriffe dürfen niemals gleichzeitig stattfinden → Ausschluss von race conditions).

Ports sind persistent

Der Wert eines Ports wird solange beibehalten, bis der Wert aktualisiert wird:



Unterstützte Porttypen:

Typ	Größe in Byte	Repräsentation	Werte
CHAR	1	2^{er} Komplement, little-endian	$-2^7 ... 2^7 - 1$
UChar	1	little-endian	$0 ... 2^8 - 1$
BOOL	1	*false* $\equiv 0$	*false*, *true*
INT16	2	2^{er}Komplement, little-endian	$-2^15 ... 2^15 - 1$
INT32	4	2^{er}Komplement, little-endian	$-2^31 ... 2^31 - 1$
INT64	8	2^{er}Komplement, little-endian	$-2^63 ... 2^63 - 1$
UINT16	2	little-endian	$0 ... 2^16 - 1$
UINT32	4	little-endian	$0 ... 2^32 - 1$
UINT64	8	little-endian	$0 ... 2^64 - 1$
FLOAT32	4	IEEE 754 single precision standard little endian	
FLOAT64	8	IEEE 754 double precision standard little endian	

Zudem werden auch Array statischer Größe von diesen Typen unterstützt.

Ports im Anwendungsbeispiel

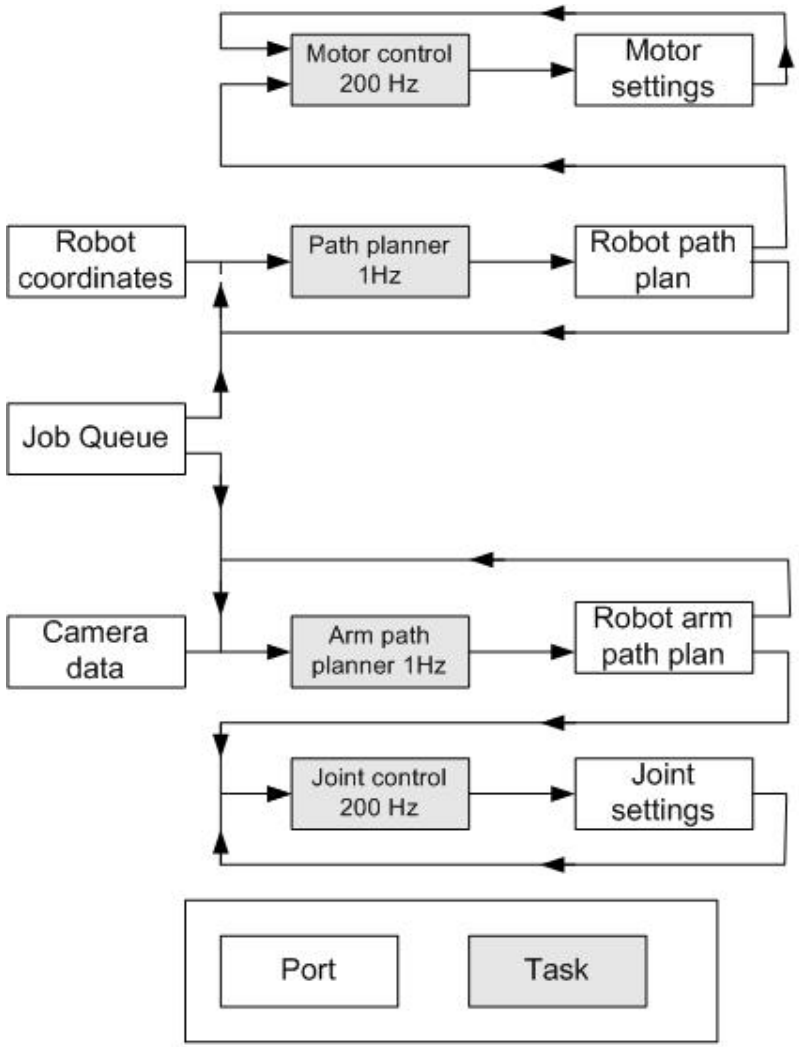
Es werden Speicherplätze für die Ergebnisse der Tasks benötigt:

- Roboterpfad
- Armpfad
- Motorenansteuerungswerte
- Gelenkansteuerungswerte

Zudem werden Ports für die einzelnen Sensoren benötigt:

- Video
- Funk
- Aufträge

Ports im Anwendungsbeispiel



Aktoren und Sensoren

- Periodisch aufgerufene Funktionen zur Kommunikation mit der Umwelt.
- Ausführung dieser Funktionen wird als instantan betrachtet $\rightarrow$ die Funktionen müssen vernachlässigbare Ausführungszeiten besitzen.
- Die Ausführung von Sensoren erfolgt zu Beginn des Aufrufintervals, die von Aktoren zum Ende des Intervals.
- Sensoren schreiben ihre Ergebnisse in Ports.
- Aktoren agieren auf der Basis von Portwerten.
- Die Lese- und Schreibzugriffe auf Ports erfolgen wie bei Tasks zeitgesteuert und deterministisch.

Aktoren und Sensoren im Anwendungsbeispiel

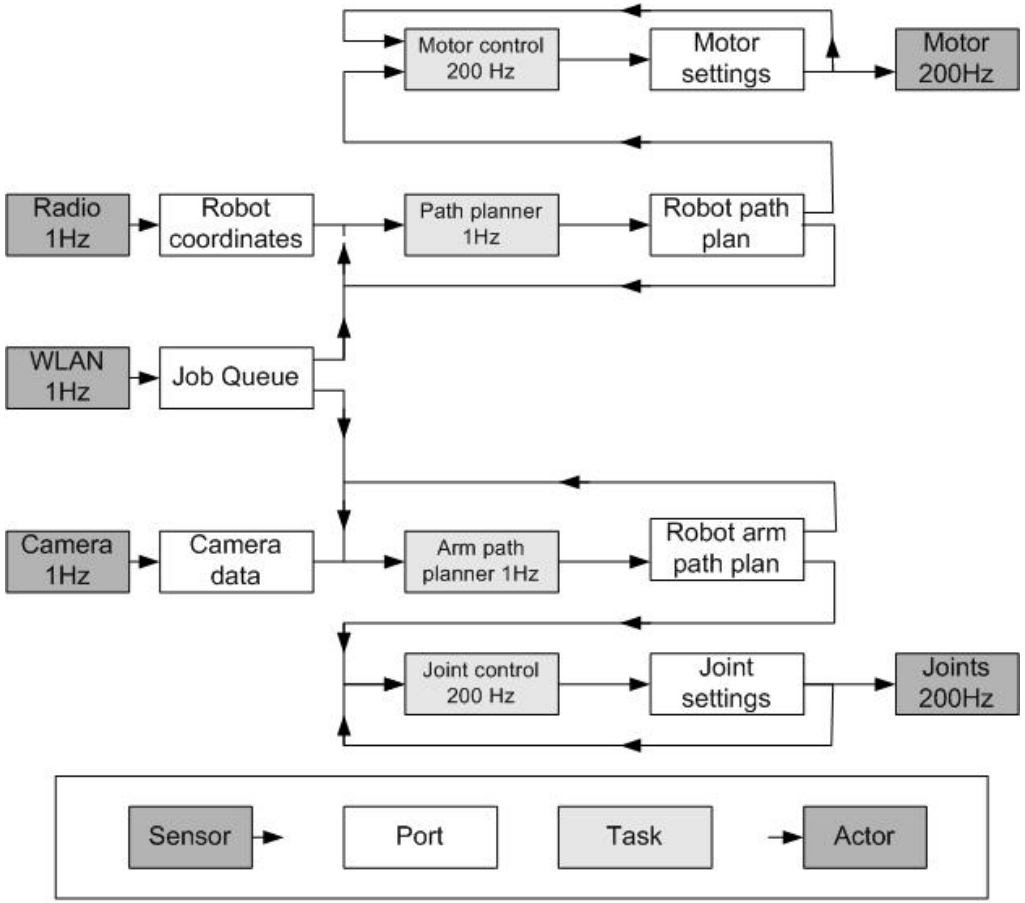
Sensoren:

- Video
- Funk
- WLAN

Aktoren:

- Motorensteuerung
- Gelenksteuerung

Aktoren und Sensoren im Anwendungsbeispiel



Modes und Guards

Zur Steuerung des Systemablaufs werden Modes, Modechanges und Guards eingeführt:

- Modes sind eine Sammlung von Tasks, Sensoren und Aktoren, die parallel auf dem System ausgeführt werden.
- Es kann immer nur ein Mode gleichzeitig ausgeführt werden.
- Der aktuelle Mode kann durch die Verwendung von Modechanges gewechselt werden.
- Ein Modechange ist eine binäre Funktion über Portwerten, die über einen Moduswechsel entscheidet.
- Guards dienen zur feineren Steuerung des Programmablaufs. Sie repräsentieren ebenfalls binäre Funktionen, die allerdings über den Start eines Tasks entscheiden.

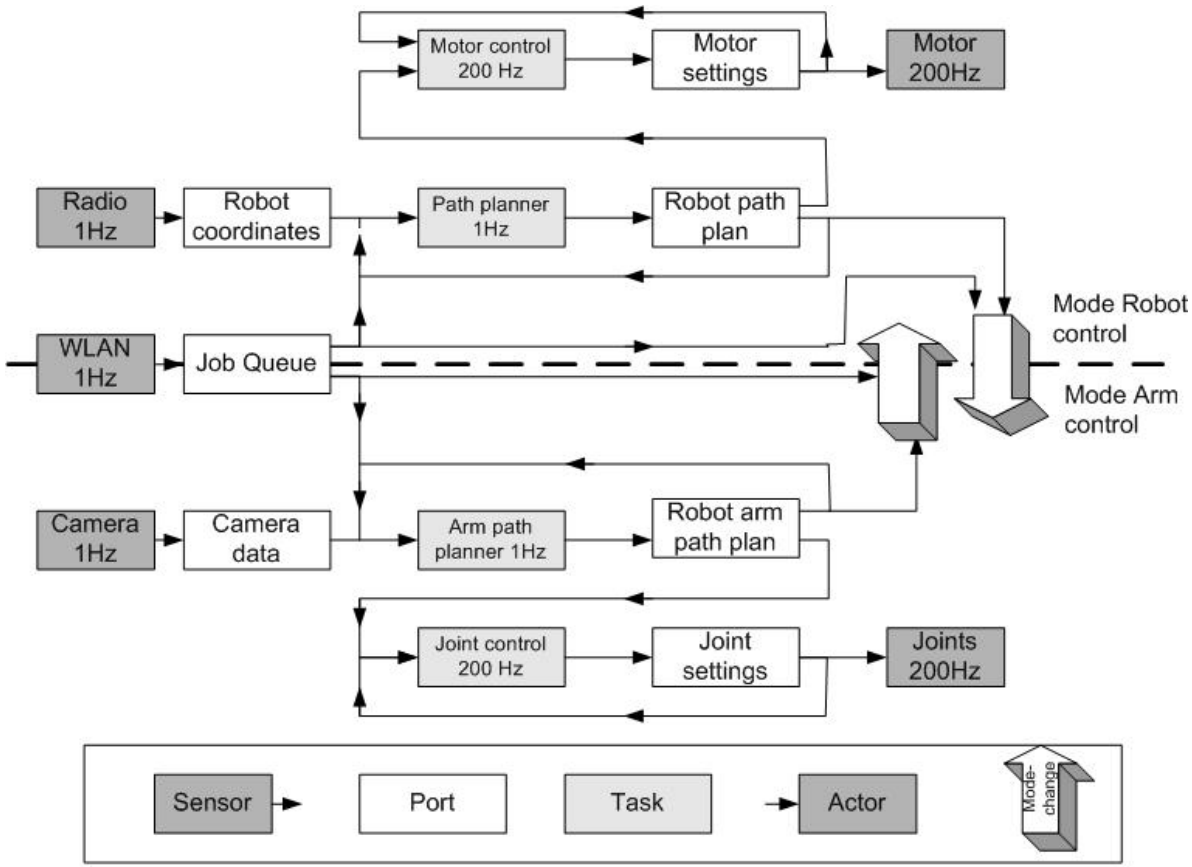
Modes im Anwendungsbeispiel

In der Anwendung kann zwischen zwei verschiedenen Modi unterschieden werden:

- Robotersteuerung
- Armsteuerung

Entsprechend werden diese Modi, sowie Moduswechsel eingeführt.

Modes im Anwendungsbeispiel



Beispiel Zerberus Code: Türsteuerung

```
/* Code for the door application*/

/*ports*/
port sens
{
        type=int;
        compareMode=NONE;
        initialValue=0;
}

port act
{
        type=int;
        compareMode=BINARY;
        initialValue=0;
}


/*actors and sensors*/
sensor input
{
        function=input();
        out=sens;
}
```

```
actor output
{
        function=output();
        in=act;
}

/*tasks*/
task manage_door
{
        function: manage_door();
        reads:sens;
        writes: act;
}

mode control_cycle
{
        startmode;
        task: manage_door 1;
        sensor: input 1;
        actor: output 1;
        duration: 10000000 ns;
}
```

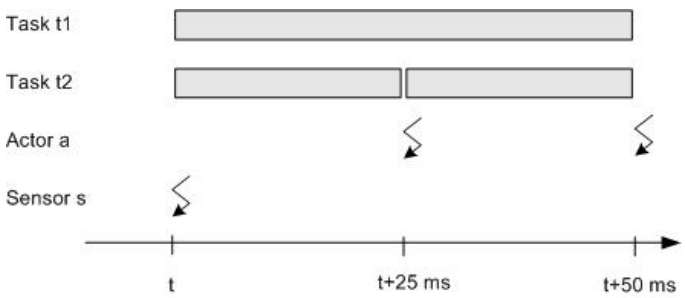
Ausführung des formalen Modells zur Laufzeit

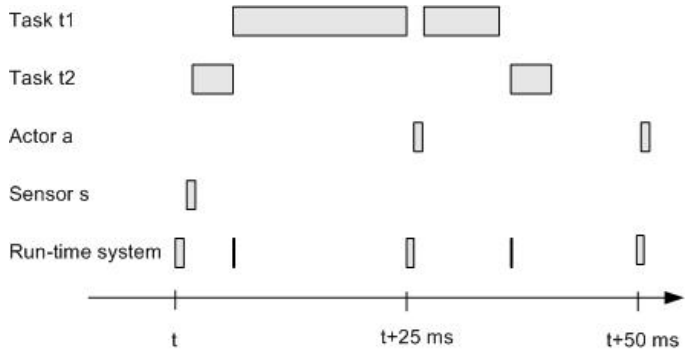
Die Ausführung des Modell erfolgt rundenbasiert.

Die Anzahl der Runden pro Moduszyklus ergibt sich aus dem kleinsten gemeinsamen Vielfachen der Frequenzen der Tasks, Sensoren und Aktoren des jeweiligen Modus.
In jeder Runde werden die folgenden Schritte durchgeführt:

1. Beenden der Tasks
2. Votierung und Synchronisation
3. Ausführung der Aktoren
4. Evaluation von Moduswechseln
5. Ausführung der Sensoren
6. Starten der Tasks
7. Warten auf nächste Runde

Formales Ausführungsmodell und tatsächliche Ausführung





Protokolle für das Laufzeitsystem

Annahmen über den Nachrichtenaustausch

Protokolle (zur Votierung, Synchronisation, Integration) erfolgen über den Nachrichtenaustausch. Zum Nachrichtenaustausch werden folgende Annahmen getroffen:

- Das Nachrichtenaustausch ist nicht unbedingt zuverlässig.
- Verfälschungen einer Nachricht werden erkannt. Die Nachricht kann allerdings verloren gehen.
- Fehlerfreie Nachrichten treffen mit bekannter maximaler Verzögerung beim Sender ein.
- Die maximale Verzögerung und damit der Synchronisationsfehler ist klein im Vergleich mit dem geforderten Regelungszeiten.

Votierung

Die Votierung erfolgt in zwei Runden:

1. In der 1. Runde werden die Zustände (Werte der Ports,aktueller Modus, aktuelle Runde) verschickt und ausgewertet.
2. In der 2. Runde werden die lokalen Sichten des Systemzustandes verschickt und der globale Zustand berechnet.

Durch die Votierung in zwei Runden können Nachrichtenverluste toleriert werden.
Alle Nachrichten sind mit der Sender-ID verknüpft (feindlicher Angriff ausgeschlossen).
Es werden nur Nachrichten berücksichtigt, die innerhalb des tolerierten Zeitintervals (Synchronisationsfehler) ankommen.

Maskierungsentscheidungen

Eine Rechner gilt als **korrekt**, wenn er mit der Mehrheit der Rechner in seinem Zustand übereinstimmt.
Jeder nicht-korrekte Rechner wird aus dem System ausgeschlossen (er darf keine Ausgaben durchführen) und startet Fehlerbehebungsmaßnahmen.
Je nach Ausgabemodell (siehe später) geben entweder alle oder der erste korrekte Rechner niedrigster ID aus.

Uhrensynchronisation

Die Uhrensynchronisation erfolgt über den Nachrichteneingang der Votierungsnachrichten.
Bei jeder empfangener und akzeptierten Nachricht wird der zeitliche Unterschied Δ_t zwischen Nachrichteneingang und erwarteten Eingang registriert.
Werden n Nachrichten empfangen, so kann der Unterschied der lokalen zur logischen globalen Uhr mit $\frac{\sum \Delta_t(i)}{n}$ abgeschätzt werden.
Übersteigt dieser Wert einen vorgegebenen Schwellwert ϵ so wird die lokale Uhr an die globale Uhr angepasst.

Integration

Über Mithören der Votierungsnachrichten wird versucht eine Integration in das laufende System zu erreichen.
Eine Integration ist jeweils immer nur zu Beginn eines Moduszyklus möglich (Zeitpunkt an dem keine Tasks am Laufen sind).
Ist eine Integration über Mithören nicht möglich, so können nötige Integrationsdaten auch mit niedrigster Priorität angefordert werden.

Problem Ausgabe: Wie kann mit Fehler bei der Ausgabe umgegangen werden?

Klassifizierung der Ausgabe:

- Votierung findet ausserhalb statt?
- Mehrfachausgabe korrekter Rechner erlaubt?
- Werden Ausgabefehler entdeckt (vom System, von der Umgebung)?

Grundsätzliches Prinzip: Votierung wird möglichst weit nach aussen verlagert, letzte Strecke wird durch sehr zuverlässige Hardware realisiert. Können Fehler bei der Ausgabe erkannt werden, so können durch Rückwärtsbehebung diese Fehler evtl. toleriert werden.

Zukünftige Aktivitäten

- Unterstützung von weiteren Fehlertoleranzmechanismen
- Trennung der Spezifikation der FT-Mechanismen von der Anwendung
- Durchführung von Pilotprojekten mit der Industrie
- Zertifizierung des Zerberus Systems

Unterstützung weiterer Fehlertoleranzmechanismen

Geforderte strukturelle Redundanz ist oftmals ein Hindernis. → andere FT-Mechanismen sollen unterstützt werden (Rückwärtsbehebung, Beschränkung der Redundanz auf einzelne Tasks).

Damit es nicht zu einer Vermischung von FT-Mechanismen und Applikation kommt, soll eine neue Sprache zur Spezifikation der angewandten Mechanismen auf dem formalen Modell eingeführt werden.

Vorgesehenes Konzept:

- **Event**: Ein Ereignis ist eingetreten, dass eine Durchführung von Fehlertoleranzmechanismen nach sich zieht (z.B. die bevorstehende Ausführung eines Actors). Auf Ereignisse kann mit angebotenen oder mit selbst implementierten FT-Mechanismen reagiert werden.
- **Exception**: Eine Ausnahme tritt auf, falls ein Fehler entdeckt wird.
- **Exception-Handling**: Auf die Entdeckung von Fehler folgt die anwendungsspezifische Fehlerbehandlung.

VL Echtzeitsysteme - Modellierung von Systemen

Prof. Alois Knoll

Lehrstuhl Embedded Systems and Robotics

Sommersemester 2005

Inhalt

- Motivation
- TLA+
- Esterel
- Real-Time UML

Literatur

- Leslie Lamport, Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers, 2002 (online unter http://research.microsoft.com/users/lamport/tla/book.html verfügbar)
- Leslie Lamport, Introduction to TLA, 1994
- Gérard Berry, Georges Gonthier, The Esterel Synchronous Programming Language: Design, Semantics, Implementation, 1992
- Bruce Powell Douglass, Doing Hard Time, 2001

Interessante Links:

- http://research.microsoft.com/users/lamport/tla/tla.html
- http://www-sop.inria.fr/meije/esterel/esterel-eng.html
- http://www.uml.org/

Modell

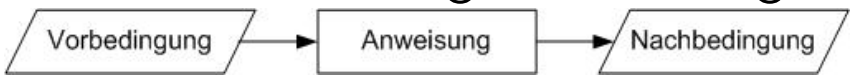
Brockhaus:
Ein Abbild der Natur unter der Hervorhebung für wesentlich erachteter Eigenschaften und Außerachtlassen als nebensächlich angesehener Aspekte. Ein M. in diesem Sinne ist ein Mittel zur Beschreibung der erfahrenen Realität, zur Bildung von Begriffen der Wirklichkeit und Grundlage von Voraussagen über künftiges Verhalten des erfaßten Erfahrungsbereichs. Es ist um so realistischer oder wirklichkeitsnäher, je konsistenter es den von ihm umfaßten Erfahrungsbereich zu deuten gestattet und je genauer seine Vorhersagen zutreffen; es ist um so mächtiger, je größer der von ihm beschriebene Erfahrungsbereich ist.

Übersicht über formale Methoden

- Deduktive (SW-)Verifikation
  - Beweissysteme, Theorem Proving
- Model Checking
  - für Systeme mit endlichem Zustandsraum
  - Anforderungsspezifikation mit temporaler Logik
- Testen
  - spielt in der Praxis eine große Rolle
  - sollte systematisch erfolgen $\longrightarrow$ ausgereifte Methodik
  - ... stets unvollständig

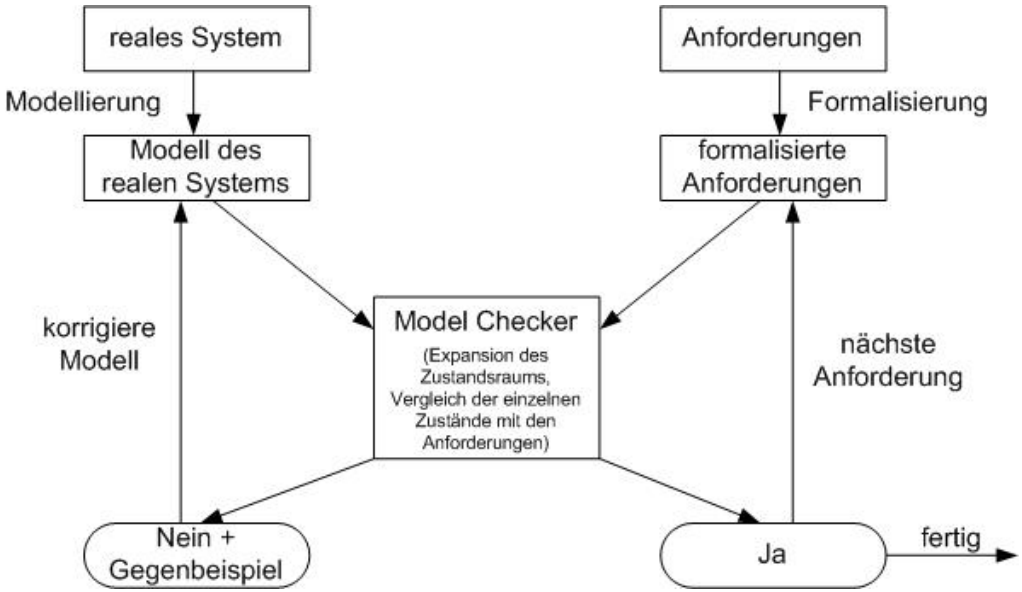
Deduktive Methoden

- Nachweis der Korrektheit eines Programms durch math.-logisches Schließen
- Anfangsbelegung des Datenraums $\longrightarrow$ Endbelegung
- Induktionsbeweise, Invarianten
  - klass. Bsp: Prädikatenkalkül von Floyd und Hoare, Betrachten von Einzelanweisungen eines Programms:



- Programmbeweise sind aufwändig, erfordern Experten
- i.A. nur kleine Programme verifizierbar
- noch nicht vollautomatisch, aber es gibt schon leistungsfähige Werkzeuge

Model Checking: prinzipielles Vorgehen



Testen

Mit Testen ist es möglich die Existenz von Fehlern nachzuweisen, nicht jedoch deren Abwesenheit

Testen ist von Natur aus unvollständig (non-exhaustive)

Es werden nur ausgewählte Testfälle / Szenarien getestet, aber niemals alle möglichen.

TLA+

TLA+

- TLA+ basiert auf TLA (Temporal Logic of Actions) und dient zur Spezifikation des funktionalen Verhalten eines Systems.
- Basiert auf Prädikatenlogik 1.Stufe
- TLA+ beschreibt die Umsetzung von TLA in ASCII-Code

Spezifikation eines Systems

Grundsätzlich wird eine **diskrete Zustandsänderung** von Computersystemen angenommen.
Ein **Verhalten (behavior)** wird formal als eine Sequenz von Zuständen spezifiziert.
Ein System kann dann durch eine Menge von solchen, möglichen **Zustandssequenzen** definiert werden. Also genau diejenigen, die eine korrekte Ausführung des Systems präsentieren.

Ein einfaches Beispiel

Ein triviales Beispiel: eine digitale Uhr, die nur die Stunde anzeigt (sehr trivial, da wir die Abhängigkeit mit der tatsächlichen Uhrzeit vernachlässigen).

Ein typisches Verhalten der Uhr wird durch folgende Zustandssequenz beschrieben:

$$[hr = 23] \rightarrow [hr = 0] \rightarrow [hr = 1] \rightarrow [hr = 2] \rightarrow ...$$

wobei $[hr = 23]$ der Zustand ist, in dem die Variable hr den Wert 23 besitzt.

TLA Code

$$
\begin{array}{rcll}
HCini \triangleq & \wedge & hr \in \{0, ..., 23\} \\
HCnxt \triangleq & \wedge & hr' = IF\ hr \neq 23\, THEN\ hr + 1\ ELSE\ 0 \\
HC \triangleq & \wedge & HCini \\
 & \wedge & \Box HCnxt
\end{array}
$$

Erklärung des Codes

HCini: HCini ist der initiale Wert der Uhr. Erlaubte Werte sind alle Zahlen zwischen 0 und 23.

HCnxt: Mit HCnxt wird die Fortentwicklung der Uhr beschrieben. Dabei wird der künftige Zustand von Variablen mit *Variablen_Name*′ (primed) gekennzeichnet. Formeln, die sowohl den alten als auch den neuen Zustand von Variablen enthalten werden als **Aktionen (actions)** bezeichnet. Eine Aktion kann in einem Schritt **wahr** oder **falsch** sein.

HC: HC ist die eigentliche Spezifikation der Uhr. Es wird gefordert, dass der initiale Wert durch HCini festgelegt ist und die Aktion HCnxt in jedem Schritt wahr ist. Diese Forderung kann durch den temporalen Logik-Operator $\Box$ spezifiziert werden.

Problem: andere Variablen

Die Spezifikation von der Uhr ist nur dann korrekt, wenn die Uhr in absoluter Isolation betrachtet wird. Ist unsere Uhr aber Teil eines größeren Subsystems, so müssen auch Schritte erlaubt sein, die es erlauben, dass der Wert der Uhr unverändert bleibt.

Beispiel: Eine Wetterstation, die neben der Uhrzeit auch die Temperatur anzeigt. Eine mögliche Zustandssequenz wäre dann:

$$\begin{bmatrix} hr = 18 \\ temp = 23.4 \end{bmatrix} \rightarrow \begin{bmatrix} hr = 19 \\ temp = 23.4 \end{bmatrix} \rightarrow \begin{bmatrix} hr = 19 \\ temp = 23.3 \end{bmatrix} \rightarrow \begin{bmatrix} hr = 19 \\ temp = 23.3 \end{bmatrix} \rightarrow \ldots$$

Lösung: stotternde Schritte (stuttering steps)

Lösungsansatz: Es müssen in der Spezifikation auch Schritte erlaubt sein, die die Uhrzeit unverändert lassen. Solche Schritte werden als **stotternde Schritte (stuttering steps)** bezeichnet.

Syntax:

$$\Box[A]_x \triangleq \Box(A \vee (x' = x))$$

Diese Formel bedeutet, dass in jedem Schritt die Aktion A wahr ist, oder aber x unverändert bleibt.

Weak Fairness

Ein Problem bei der Spezifikation der Uhr bleibt: durch die Einführung von stotternden Schritten sind auch Zustandssequenzen erlaubt, in denen die Uhr stehen bleibt.

Forderung: Es muss eine nicht endliche Anzahl an nicht stotternden Schritten geben.

Lösung: Weak Fairness: ist die Aktion möglich, so soll sie in unendlich oft ausgeführt werden. Eine Aktion A in einem Zustand s ist möglich (**enabled**), wenn es ein Zustandspaar (s', s), so dass s' der Nachfolgezustand von s ist und die Aktion A wahr ist.

Syntax:

$$WF_x[A]$$

Bedeutung: Falls $A \wedge (x' \neq x)$ irgendwann möglich wird und für immer möglich bleibt, wird die Aktion A unendlich oft ausgeführt.

Endgültiger TLA Code

$$
\begin{aligned}
HCini \triangleq\ & \wedge\ hr \in \{0, ..., 23\} \\
HCnxt \triangleq\ & \wedge\ hr' = IF\ hr \neq 23\, THEN\ hr + 1\ ELSE\ 0 \\
HC \triangleq\ & \wedge\ HCini \\
& \wedge\ \Box[HCnxt]_{hr} \\
& \wedge\ WF_{hr}[HCnxt]
\end{aligned}
$$

Was bedeuten TLA Formeln?

- Eine TLA Formel ist wahr oder falsch in Bezug auf ein **Verhalten (behavior)**.
- Ein Verhalten ist eine Sequenz von **Zuständen**.
- Ein Zustand weist den Variablen einen Wert zu.
- Kann nun für ein System ein Verhalten (in Form von Zustandssequenzen) gefunden werden, so kann eine Aussage getroffen werden, ob das System eine Formel erfüllt.

Zusammenfassung von TLA

P	Die Formel wird durch ein Verhalten erfüllt, falls P für den initialen Zustand wahr ist.
$\Box [A]_f$	Erfüllt durch ein Verhalten, falls in jedem Schritt A wahr ist oder f unverändert bleibt.
$\Box [A]$	Erfüllt durch ein Verhalten, falls A in jedem Schritt wahr ist.
$\exists x : F$	Erfüllt durch ein Verhalten, falls es einen Wert gibt durch dessen Zuweisung an x ein Verhalten erzeugt werden kann, dass F erfüllt.
$WF_f(A)$	Erfüllt durch ein Verhalten, falls $A \wedge f' \neq f$ unendlich oft nicht möglich ist oder unendlich viele $A \wedge f' \neq f$ Schritte vorkommen.
$SF_f(A)$	Erfüllt durch ein Verhalten, falls $A \wedge f' \neq f$ nur endlich oft möglich ist oder unendlich viele $A \wedge f' \neq f$ Schritte vorkommen.
$F \xrightarrow{+} G$	Die Formel ist durch ein Verhalten erfüllt, falls G mindestens solange für das Verhalten erfüllt ist wie F.
$\Diamond F$	Definiert als $\neq \Box \neq F$ (eventuell wahr).
$F \rightsquigarrow G$	Definiert als $\Box F \Longrightarrow \Diamond G$ (jedes Mal wenn F wahr wird, ist G eventuell wahr)

Spezifikation in TLA+

TLA+ beschreibt die Umsetzung von Formeln in ASCII-Code:

- Reservierte Wörter werden durch Großbuchstaben gekennzeichnet (z.B. EXTENDS).
- Falls möglich werden Symbole bildhaft in ASCII geschrieben (z.B. [](□), # (≠)).
- Falls keine gute ASCII-Repräsentation möglich ist, wird die LATEX-Notation benutzt. z.B. ∈ \in

Code in TLA+

```
---------------Module HourClock ---------------
EXTENDS Naturals
VARIABLE hr
HCini == hr \in (0..23)
HCnxt == hr' = IF hr # 23 THEN hr+1 ELSE 0
HC == HCini /\ [] [HCnxt]_hr
-----------------------------------------------
THEOREM HC => [] HCini
===============================================
```

Echtzeitfähigkeit

Schwache oder starke (siehe später) Fairness reichen aber noch nicht aus um Antwortzeiten zu garantieren.

TLA benutzt zur Referenzierung der Uhrzeit die Variable *now*. Die übliche Einheit für *now* ist Sekunden. Die Zustände der Uhr werden als diskret angenommen, die Granularität ist nicht näher spezifiziert.

Aktionen werden als augenblicklich (**instantaneous**) angenommen. Die Ausführung dauert also keine Zeit.

Beispiel Echtzeituhr

```
------------------------ MODULE RealTimeHourClock ------------------------
EXTENDS Reals, HourClock
VARIABLE now
CONSTANT Rho
ASSUME (Rho \in Real) /\  (Rho > 0)
--------------------------------------------------------------------------

   ------------------------- MODULE Inner --------------------------------
   VARIABLE t
   TNext == t' = IF HCnxt THEN 0 ELSE t+(now'-now)
   Timer   == (t = 0)  /\  [][TNext]_<<t,hr, now>>
   MaxTime == [](t \leq  3600 + Rho)
   MinTime == [][HCnxt => t \geq 3600 - Rho]_hr
   HCTime  == Timer /\ MaxTime /\ MinTime
  ==========================================================================

I(t) == INSTANCE Inner

NowNext == /\ now' \in {r \in Real : r > now}
           /\ UNCHANGED hr

RTnow == /\ now \in Real
         /\ [][NowNext]_now
         /\ \A r \in Real : WF_now(NowNext /\ (now'>r))

RTHC == HC  /\  RTnow /\  (\EE t : I(t)!HCTime)
==========================================================================
```

Erläuterung

TNext	Der interne Timer wird immer um die vergangene Zeit erhöht. Wird die Uhr erhöht, so wird der Timer wieder zurückgesetzt.
Timer	Formel für die Uhr mit Initialisierung des internen Timers
MaxTime	Erlaubt ist ein Drift um maximal Rho Sekunden $\rightarrow$ nach 3600 + Rho Sekunden muss die Uhr spätestens fortgeschaltet werden.
MinTime	Zweite Driftbedingung: die Uhr darf frühestens nach 3600 - Rho Sekunden fortgeschaltet werden
HCTime	Spezifikation der Echtzeituhr (allerdings noch ohne genauere Spezifikation des Zeitfortschritts)
NowNext	Die Uhrzeit erhöht sich immer
RTnow	Es gibt immer einen Zeitpunkt, der später ist als jeder beliebige Zeitpunkt.
RTHC	Endgültige Spezifikation

Werkzeuge für TLA+

- Syntaxanalyseprogramm
- TLATEX(Schriftsatzprogramm)
- TLC Modellchecker

TLC Model Checker

Der Model Checker behandelt Spezifikationen in der Standardform:

$$Init \wedge \Box[Next]_{vars} \wedge Temporal$$

wobei Init das initiale Prädikat, Next die Aktionen, vars das Tupel aller Variablen und Temporal die temporalen Formeln, die typischerweise die Lebendigkeit (liveness) spezifizieren, beschreiben.

Einschränkung: der temporale Existenzquantor wird nicht behandelt.

Testmöglichkeiten: TLC erlaubt:

- die Prüfung, ob eine Spezifikation Anforderung in der Form von Formeln erfüllt.
- die Entdeckung von Leichtsinnsfehler, z.B. $3+ < 1, 2 >$
- die Entdeckung von Deadlocks. Der Ausschluss eines Deadlocks kann durch die Eigenschaft $\Box(ENABLEDNext)$ ausgedrückt werden.

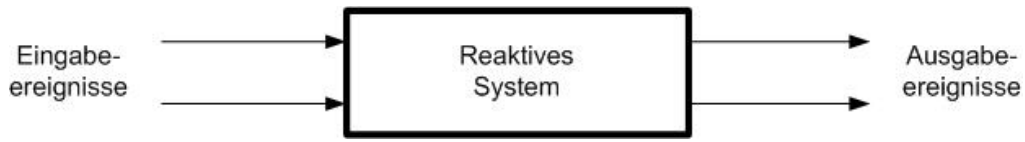
Betriebsmodi des TLC

1. **Modell Checking:** Es werden alle erreichbaren Zustände gesucht. Durch die Angabe von Restriktionen durch den Benutzer kann die Anzahl der erreichbaren Zustände reduziert werden. Für alle Zustände werden die gestellten Anforderungen geprüft. Der Algorithmus startet bei den initialen Zuständen und sucht dann nach möglichen Nachfolgezuständen.
2. **Simulation:** Zufällige Erzeugung von Verhalten (Zustandssequenzen) und Tests dieser Verhalten. Es wird dabei nicht versucht alle erreichbaren Zustände abzudecken. Der Benutzer muss eine maximale Länge der Zustandssequenzen festlegen. Der Algorithmus startet jeweils zufällig mit einem initialen Zustand und wählt einen beliebigen Nachfolgezustand. Für jeden dieser Zustände werden die Anforderungen geprüft. Die Simulation wird durch den Entwickler abgebrochen.

Esterel

Einführung in Esterel

- Eingeführt durch G. Berry
- Dient zur Beschreibung von reaktiven Systemen
- Esterel gehört zu der Familie der synchronen Sprachen, weitere Vertreter: Lustre, Signal, Statecharts



Synchrony hypothesis

Die Synchronitätshypothese (**synchrony hypothesis**) nimmt an, dass die zugrundeliegende physikalische Maschine des Systems unendlich schnell ist.

$\rightarrow$ Die Reaktion des Systems auf ein Eingabeereignis erfolgt augenblicklich. Reaktionsintervale reduzieren sich zu Reaktionsmomenten (reaction instants).

Rechtfertigung: Diese Annahme ist korrekt, wenn die Wahrscheinlichkeit des Eintreffens eines zweiten Ereignisses, während der initialen Reaktion auf das vorangegange Ereignis, sehr klein ist.

Esterel erlaubt das gleichzeitige Auftreten von mehreren Eingabeereignissen. Der Reaktionsmoment ist in Esterel dann komplettiert, wenn das System auf alle Ereignisse reagiert hat.

Determinismus

Esterel setzt den Determinismus der Anwendung voraus: auf eine Sequenz von Ereignissen (auch gleichzeitigen) muss immer dieselbe Sequenz von Ausgabeereignissen folgen.

Alle Esterel Anweisungen und Konstrukte sind garantiert deterministisch. Die Forderung nach Determinismus wird durch den Esterel Compiler überprüft.

Sprache Esterel

Parallelismus

Zur parallelen Komposition stellt Esterel den Operator || zur Verfügung. Sind **P1** und **P2** zwei Esterel Programme, so ist auch **P1**||**P2** ein Esterel Programm mit folgenden Eigenschaften:

- Alle Eingabeereignisse stehen sowohl **P1** als auch **P2** zur Verfügung.
- Jede Ausgabe von **P1** (oder**P2**) ist im gleichen Moment für **P2** (oder **P1**) sichtbar.
- Sowohl **P1** als auch **P2** werden parallel ausgeführt und die Anweisung **P1**||**P2** endet erst, wenn beide Programme beendet sind.
- Es könnnen keine Daten oder Variablen von **P1** und **P2** gemeinsam genutzt werden.

Module

Module definieren in Esterel (wiederverwendbaren) Code. Module haben ähnlich wie Subroutinen ihre eigenen Daten und ihr eigenes Verhalten.

Allerdings werden Module nicht aufgerufen, vielmehr findet eine Ersetzung des Aufrufs durch den Modulcode zur Compile-Zeit statt.

Globale Daten werden nicht unterstützt. Ebenso sind rekursive Moduldefinitionen nicht erlaubt.

Syntax:

%this is a line comment
module module-name:
declarations and compiler directives
%signals, local variables etc.
body
.% end of module body

Zeitdauer

Die Zeitachse wird in Esterel in diskrete **Momente (instants)** aufgeteilt. Über die Granularität wird dabei in Esterel keine Aussage getroffen.

Zur deterministischen Vorhersage des zeitlichen Ablaufes von Programmen wird jede Anweisung in Esterel mit einer genauen Definition der Ausführungszeitdauer verknüpft.
So terminiert beispielsweise **emit** sofort, während **await** so viel Zeit benötigt, bis das assoziierte Signal verfügbar ist.

Signale

Zur Modellierung der Kommunikation zwischen Komponenten (Modulen) werden **Signale** eingeführt. Signale sind eine logische Einheit zum Informationsaustausch und zur Interaktion.

Deklaration Die Deklaration eines Signals erfolgt am Beginn des Moduls. Der Signalname wird dabei typischerweise in Großbuchstaben geschrieben. Zudem muss der Signaltyp festgelegt werden.

Esterel stellt verschiedene Signale zur Verfügung. Die Klassifikation erfolgt nach:

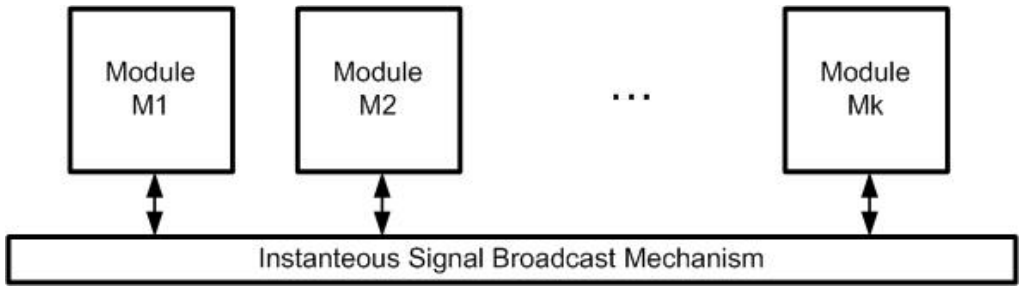
- Sichtbarkeit: Schnittstellen (interface) Signale vs. lokale Signale
- enthaltener Information: pure Signale vs. wertbehaftete Signale (typisiert)
- Zugreifbarkeit der Schnittstellensignale: Eingabe (input), Ausgabe (output), Ein- und Ausgabe (inputoutput), Sensor (Signal, dass immer verfügbar ist und das nur über den Wert zugreifbar ist)

Signal Broadcast Mechanismus

Versand: Der Versand von Signalen durch die **emit** Anweisung erfolgt über einen Broadcast Mechanismus. Signale sind immer sofort für alle anderen Module verfügbar.

Verfügbarkeit: Signale sind nur für den bestimmten Moment verfügbar. Nach Ende des aktuellen Moments wird der Bus zurückgesetzt.

Zugriff: Prozesse können per **await** auf Signale warten oder prüfen, ob ein Signal momentan vorhanden ist (**present**). Auf den Wert eines wertbehaftete Signale kann mittels des Zugriffsoperator **?** zugegriffen werden.



Ereignisse (Events)

Ereignisse setzen sich zu einem bestimmten Zeitpunkt (instant) aus den Eingabesignalen aus der Umwelt und den Signalen, die durch das System als Reaktion ausgesandt werden, zusammen.

Esterel Programme können nicht direkt auf das ehemalige oder zukünftige Auftreten von Signalen zurückgreifen. Auch kann nicht auf einen ehemaligen oder zukünftigen Moment zugegriffen werden.

Beispiel: Kaffee- und Teeautomat

```
module VM1:
input COIN, TEA, COFFEE;
output SERVE_TEA, SERVE_COFFEE;
relation COIN # TEA # COFFEE;
loop
    await COIN;
    await
        case TEA do emit SERVE_TEA;
        case COFFEE do emit SERVE_COFFEE;
    end await;
end loop;
.
```

Beziehungen (relations)

Um bei der automatischen Generierung des endlichen Automaten des Systems die Größe zu reduzieren, können über die **relation** Anweisung Einschränkungen in Bezug auf die Signale spezifiziert werden:

relation Master-signal-name => Slave-signal-name;

Bei jedem Auftreten des Mastersignals muss auch das Slave-Signal verfügbar sein.

relation Signal-name-1 # Signal-name-2 # ... # Signal-name-n;

In jedem Moment darf maximal eines der spezifizierten Signale Signal-name-1, Signal-name-2 ,..., Signal-name-n präsent sein.

Unendliche Schleife (infinite loop)

```
loop Body end loop;
```

Mit Hilfe dieser Anweisung wird ein Stück Code **Body** endlos ausgeführt. Sobald eine Ausführung des Codes beendet wird, wird der Code wieder neu gestartet.

Bedingung: die Ausführung des Codes darf nicht im gleichen Moment terminieren, indem sie gestartet wurde

await Anweisung

```
await
    case Occurence-1 do Body-1
    case Occurence-2 do Body-2
    ...
    case Occurence-n do Body-n
end await;
```

Mit Hilfe dieser Anweisung wird auf das Eintreten einer Bedingung gewartet. Im Falle eines Auftretens wird der assoziierte Code gestartet. Werden in einem Moment mehrere Bedingungen wahr, entscheidet die textuelle Reihenfolge. So kann eine deterministische Ausführung garantiert werden.

emit Anweisung

```
emit Signal_Name;
```

Mit Hilfe dieser Anweisung wird ein Signal über den Broadcast-Mechanismus gesendet. Die emit-Anweisung terminiert augenblicklich $\rightarrow$ das Signal ist im gleichen Moment verfügbar.

Testen von Programmen

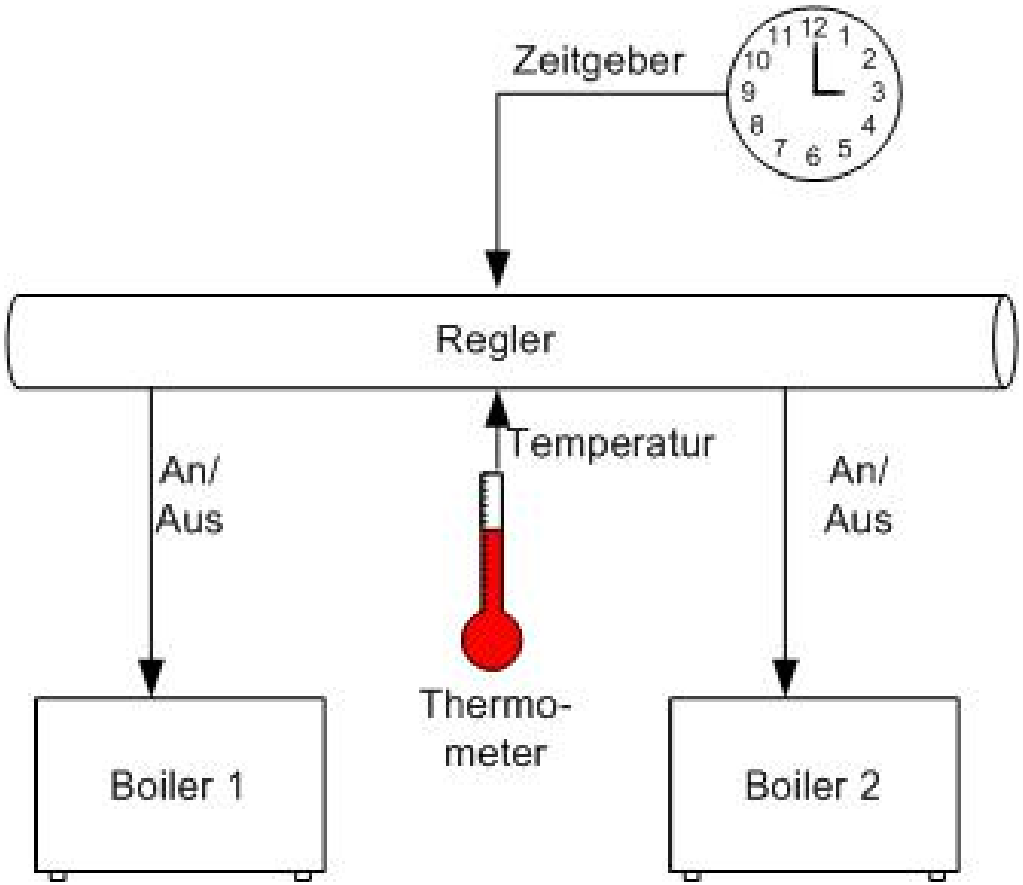
Programme können durch die Verwendung von Testfällen getestet werden:

1. T1=({COIN},{}),({TEA},{SERVE_TEA})
2. T2=({COIN},{}),({COFFEE},{SERVE_COFFEE})
3. T3=({COIN},{}),({COIN},{}),({TEA},{SERVE_TEA}),
({TEA},{SERVE_TEA})

T1 und T2 sind erfolgreich, T3 nicht

Form: Ein Testfall besteht aus einer Listen von Momenten, wobei jeder Moment durch die Eingabe- und Ausgabeereignisse charakterisiert wird.

Weiteres Beispiel: Temperaturregelung



Beschreibung Beispiel

Ziel: Regelung der Temperatur (Zielwert: 250 Grad Celsius)

Ansatz: Fällt die Temperatur unter den Zielwert, so wird zunächst Boiler 1 eingeschaltet. Wird die Zieltemperatur nach einer gewissen Zeitdauer Delta_T nicht erreicht, so wird ein zweiter Boiler hinzugeschaltet. Steigt die Temperatur über den Zielwert, so werden alle Boiler abgeschaltet.

Code 1. Teil

```
module temp_controller:
input TEMP:integer, SampleTime, Delta_T;
output B1_ON,B1_OFF, B2_ON,B2_OFF;
relation SAMPLE_TIME => TEMP;
signal HIGH, LOW in
    every SAMPLE_TIME do
        present TEMP else await TEMP end present;
        if(?TEMP>=250) then emit HIGH else emit LOW end if;
    end every;
```

Code 2. Teil

```
    ||
    loop
        await LOW;
        emit B1_ON;
        do
            await HIGH;
            emit B1_OFF;
        watching Delta_T
        timeout
            present HIGH else
                emit B2_ON
                await HIGH;
                emit B2_OFF
            end present;
            emit B1_OFF;
        end;
    end loop;
end;
.
```

Lokale Signale und wertbehaftete Signale

```
signal Signal-decl-1,Signal-decl-2,...,Signal-decl-n in
    Body
end;
```

Durch diese Anweisung werden lokale Signale erzeugt, die nur innerhalb des mit Body bezeichneten Code verfügbar sind.

```
Signal-name: Signal-type
```

Der Typ eines wertbehaften Signals kann durch diese Konstruktion spezifiziert werden.

do-watching-timeout-Konstrukt

Das do-watching-timeout Konstrukt ist eine grundsätzliche und wichtige zeitliche Anweisung.

Syntax:

```
do
    Body-1
watching Occurence
timeout
    Body-2
end;
```

Semantik: Body-1 wird zunächst gestartet. Endet Body-1 vor dem Ablauf des Timeouts Occurence, so ist die do-watching-timeout Anweisung zu Ende. Läuft der Timeout vor der Beendigung von Body-1 aus, so wird die Ausführung von Body-1 augenblicklich gestoppt und stattdessen der Code Body-2 ausgeführt.

every Anweisung

Mit Hilfe der every Anweisung kann ein periodisches Wiederstarten implementiert werden.

Syntax:

```
every Occurence do
    Body
end every;
```

Semantik: Jedes Mal falls die Bedingung Occurence erfüllt ist, wird der Code Body gestartet. Falls die nächste Bedingung Occurence vor der Beendigung der Ausführung von Body auftritt, wird die aktuelle Ausführung sofort beendet und eine neue Ausführung gestartet.
Es ist auch möglich eine Aktion in jedem Moment zu starten:

```
every Tick do
    Body
end every;
```

present Anweisung

Durch Verwendung der present Anweisung kann die Existenz eines Signals geprüft werden.

Syntax:

```
present Signal-Name then
    Body-1
else
    Body-2
end present;
```

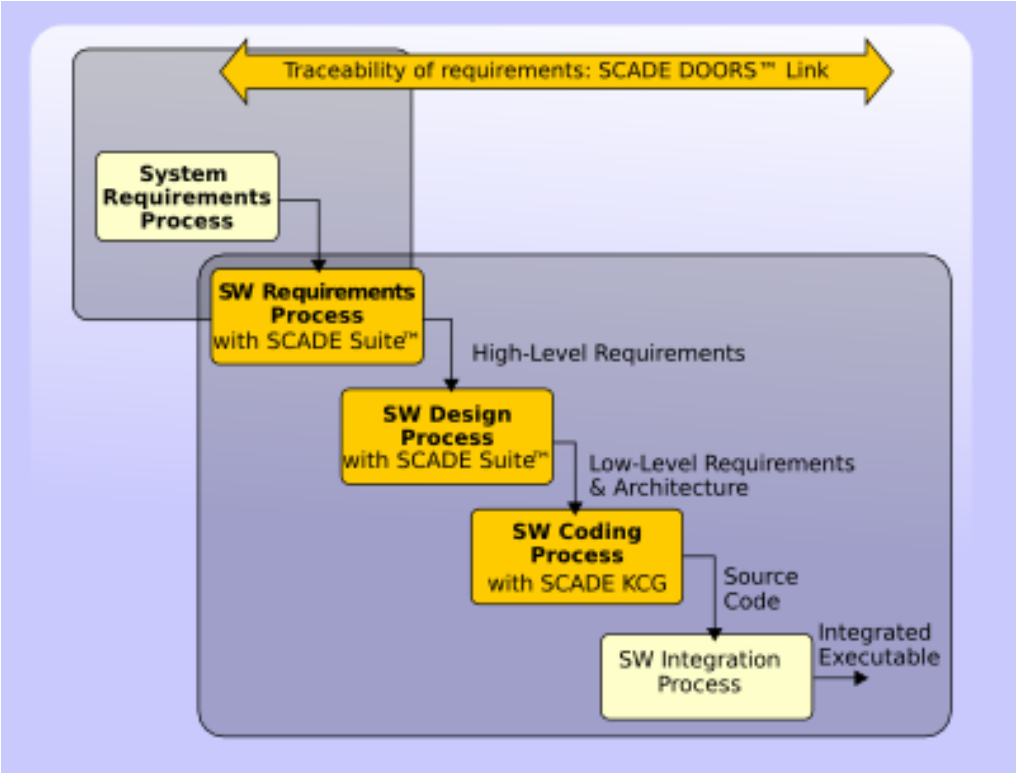
Semantik: Bei Start dieser Anweisung wird geprüft, ob das Signal Signal-Name verfügbar ist. Ist es verfügbar, so wird der Code von Body-1 ausgeführt, anderenfalls von Body-2. Innerhalb der present-Anweisung kann auch entweder der **then Body-1** oder der **else Body-2** -Teil weggelassen werden.

Werkzeuge für Esterel

Für Esterel stehen viele Werkzeuge zur Verfügung:

- Compiler
- Codegeneratoren
- Werkzeuge zur Simulation
- Theorembeweiser
- Automatenvisualisierung

Entwicklungsprozess mit Esterel



Real-Time UML

Wieso UML

- UML Standard in Entwicklungsprozessen von Standardsoftware
- Breite Unterstützung durch Tools (Hersteller: Artisan, Rationale, I-Logix)
- Intuitiver Ansatz

Diagrammtypen (Beispiel Artisan)

Diagrammtyp	**Zweck**
Aktivitätsdiagramme Activity D. (UML)	Zeigt Aktionssequenzen in verschiedenen Kontexten
Klassendiagramme Class D. (UML)	Zeigen Pakete und Paketabhängigkeiten, Klassen und deren Eigenschaften und Beziehungen
Nebenläufigkeitsdiagramme Concurrency D. (Artisan)	Modelliert Prozesse und Prozessinteraktions-mechanismen
Einschränkungsdiagramme Constraints D. (Artisan)	Spezifiziert nicht-funktionale Systemanforderungen
Objektinteraktionsdiagramme Object Collaboration D. (UML)	Zeigt Objektinteraktionen für spezielle Anwendungs-fälle oder für den allgemeinen Fall
Objektsequenzdiagramme Object Sequence D. (UML)	Zeigt Subsystem/Gerät/Objekt/Prozess-Interaktionen für einen kompletten Testfall als Sequenz von Systemnachrichten
Zustandsdiagramme State D. (UML)	Beschreibt zustandsbasiert das dynamische Verhalten von Klassen oder Subsystemen

Diagrammtypen Fortsetzung

Strukturdiagramm Structure D. (UML)	Modelliert die Zusammensetzung von strukturierten Klassen durch die Benutzung von Teilen, Ports, Schnittstellen und IO-Flüssen
Systemarchitekturdiagramm System Architecture D. (Artisan)	Zeit Systemhardwarekomponenten, deren Zusammenhänge und Eigenschaften
Beziehungstabellendiagramme Table Relationships D. (Artisan)	Spezifiziert die Beziehungen zwischen persistenten Dateneinheiten
Anwendungsfalldiagramme Use Case D. (UML)	Identifiziert Systemdienste in Form einer funktionellen Beschreibung und verbindet diese mit externen Aktoren
Generelles Grafikdiagramm General Graphics D. (Artisan)	Unterstützt nicht-spezifische Diagrammmöglichkeiten
Text-Diagramm Text D. (Artisan)	Untertützt die Möglchkeit zur Speicherung von Textdokumenten in einem Modell

Erweiterungen der Standardmodelle

UML ist in seiner Grundform nicht für Echtzeitsysteme geeignet. Die OMG (object management group) führte deshalb als Erweiterung sogenannte **Stereotypen (stereotypes)** ein.

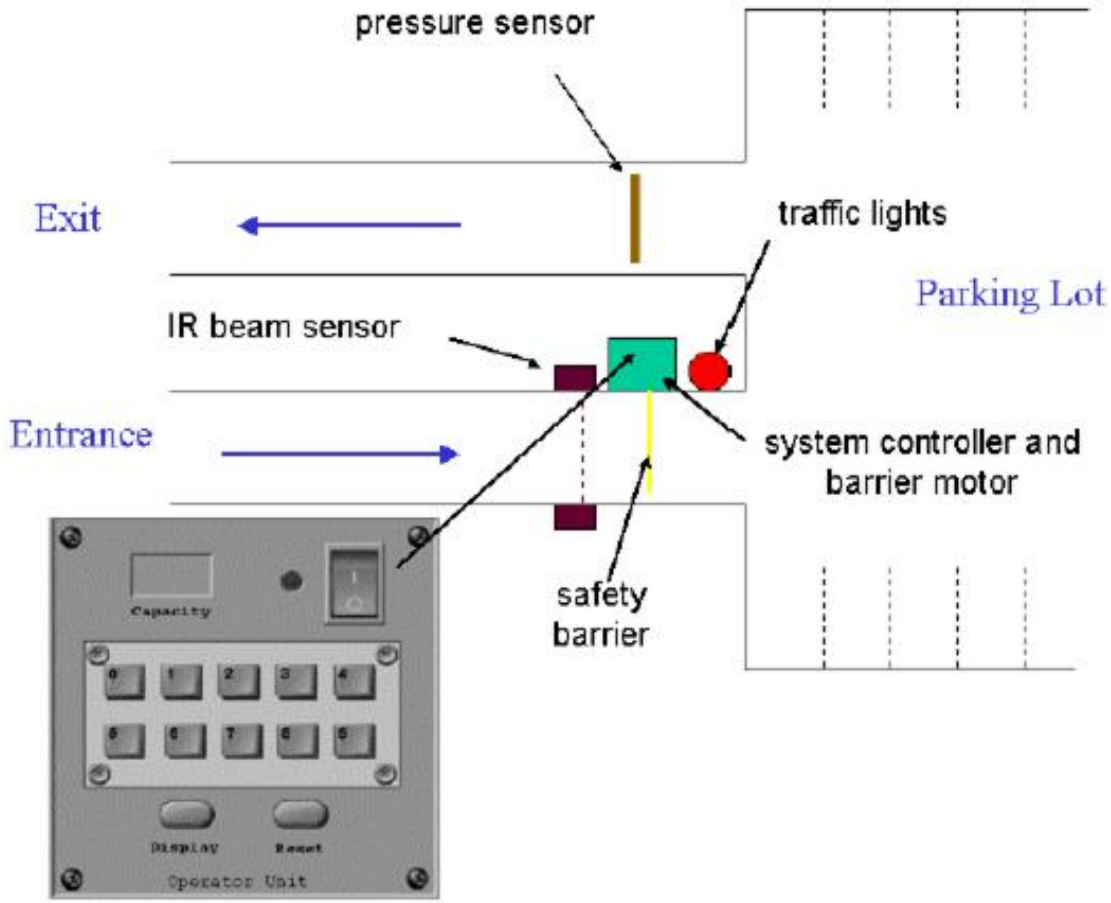
Stereotypen erlauben die Erzeugung von neuen Meta-Modellen der UML-Modelle.

Beispiel: Mit dem Stereotyp <<active>> kann eine Klasse markiert werden, die die Basis für einen eigenständigen Prozess ist

Beispiele: UML-Stereotypen für Echtzeitsysteme

UML Basistyp	**Stereotyp**	**Beschreibung**
Class	<<active>>	UML Klasse ist Basis für einen Thread
Message	<<synchronous>>	Assoziation wird als einfache Funktion oder Methodenaufruf realisiert
	<<blocking-local>>	Assoziation überschreitet Thread-Grenze, aber der aufrufende Thread wird bis zur Aufruf -rückkehr blockiert
	<<asynchronous-local>>	Assoziation überschreitet Thread-Grenze durch Senden der Nachricht an die Eingangs-nachrichtenwarteschlange des Zielthreads
	<<waiting-local>>	Sender wartet auf Empfänger mit maximaler Wartezeit
	<<synchronous-remote>>	Assoziation überschreitet Prozessorgrenze und Sender wartet auf Empfänger
...	...	...
...	...	...

Anwendungsbeispiel: Parkanlage



Beschreibung von Beispiel

- Parkanlage regelt die Einfahrt in die Parkgarage
- Freie Parkplätze werden kontrolliert und Autos werden nur in die Parkgarage gelassen, falls noch ein Parkplatz frei ist
- Anlage verwendet zwei Sensoren: Infrarotsensor am Eingang, Drucksensor am Ausgang
- Durch eine Schranke werden Autos gehindert in die Parkgarage einzufahren
- Ein rotes und ein grünes Licht signalisieren dem Fahrer, ob Plätze frei sind
- Über eine Bedieneinheit kann die Kapazität angepasst werden.

Erklärung wichtiger Diagrammtypen am Beispiel

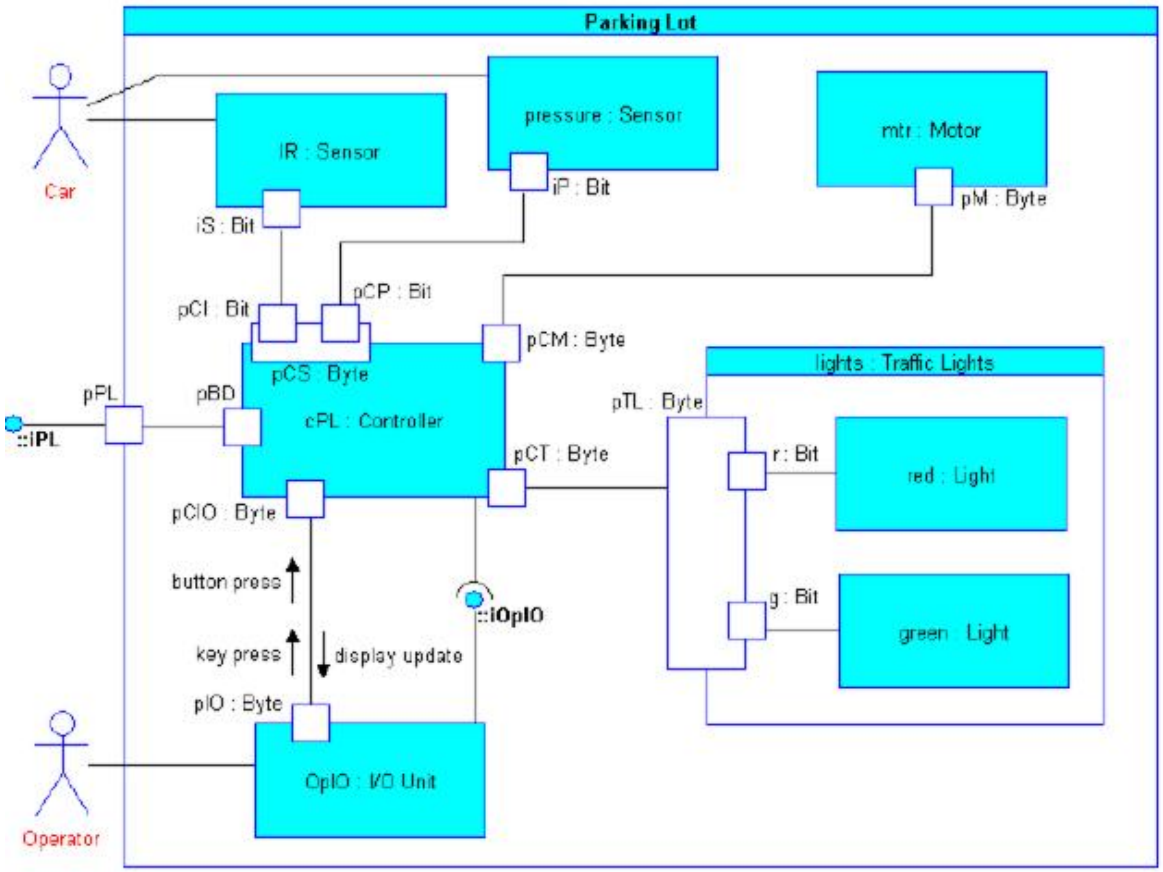
Structure Diagrams

Ziele:

- Aufteilung des Systems in seine Subsysteme und Komponenten
- Identifikation der Kommunikation
- Identifikation der Speichertypen
- Identifikation der Komponententypen
- Spezifikation der Schnittstellen

Zeitpunkt: Analysephase

Structure Diagrams (2)



Use Cases

Ziele:

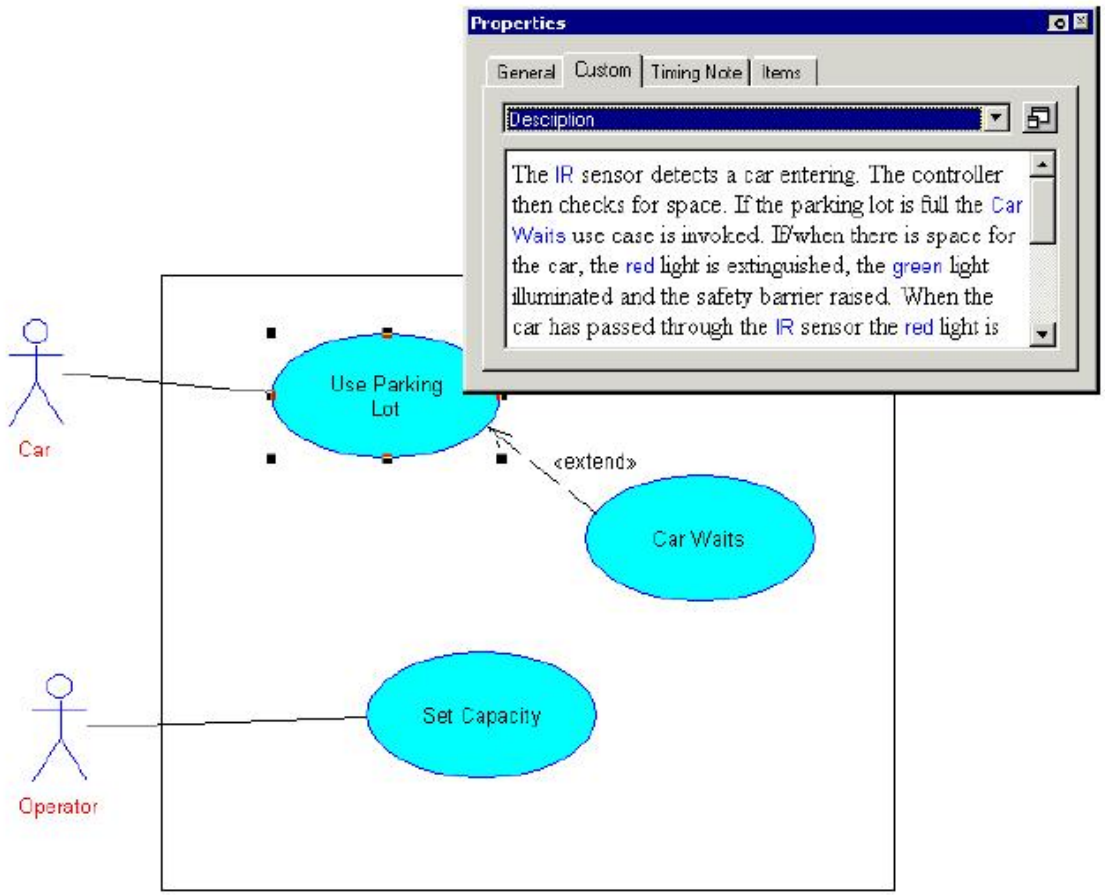
- Identifikation der Anwendungsfunktionalität durch textuelle und graphische Beschreibung

Methoden:

- Testfälle beschreiben immer nur ein Anwendungsszenario
- Testfälle beginnen immer mit einer Aktion einer externen Einheit
- Testfälle beschreiben Funktionalität in textueller Form aus Sicht des externen Benutzers

Zeitpunkt: Analysephase

Use Cases (2)


Properties
General
Custom
Timing Note
Items
Desciption
The IR sensor detects a car entering. The controller then checks for space. If the parking lot is full the Car Waits use case is invoked. If/when there is space for the car, the red light is extinguished, the green light illuminated and the safety barrier raised. When the car has passed through the IR sensor the red light is
Use Parking Lot
«extend»
Car Waits
Car
Set Capacity
Operator

Sequence diagram

Ziele:

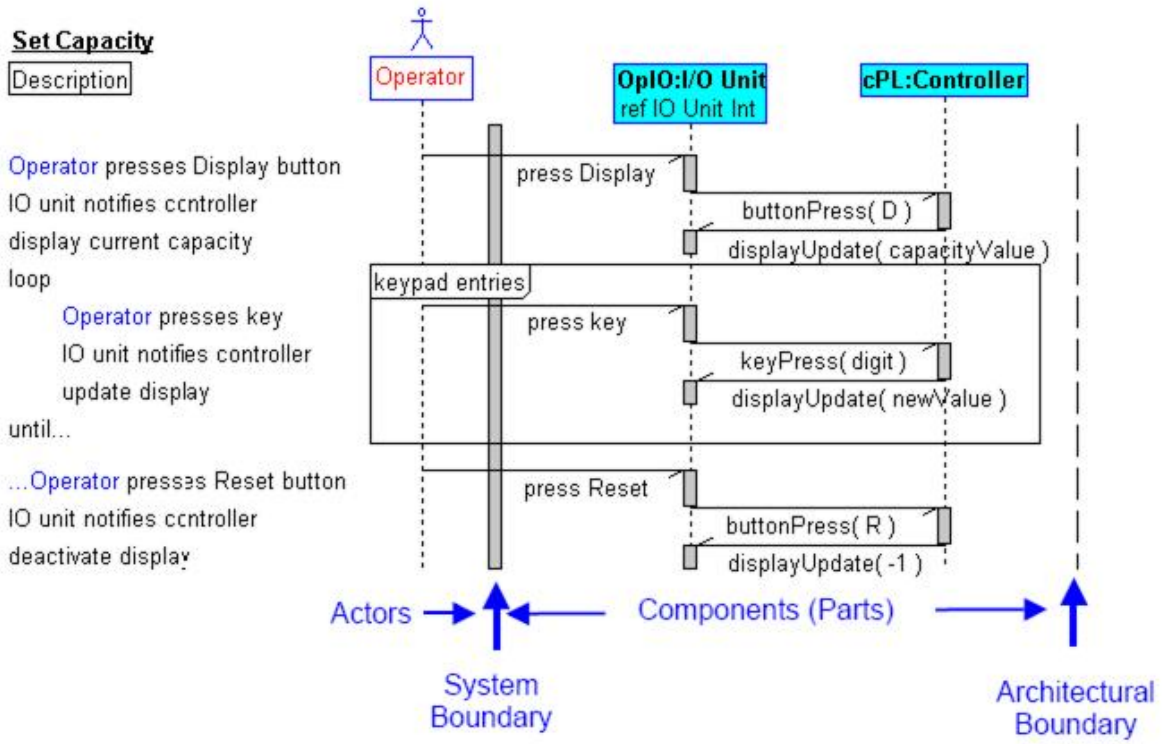
- Analyse der dynamischen Interaktion der einzelnen Komponenten

Methoden:

- Definition von Nachrichtensequenzen zwischen einzelnen Komponenten
- Sequenzdiagramme basieren zumeist auf Use Cases
- Einzelnen Komponenten werden als Black Boxes betrachtet
- Klassifikation der Kommunikation (synchron, asynchron)

Zeitpunkt: Designphase

Sequence diagram (2)



Class diagram

Ziele:

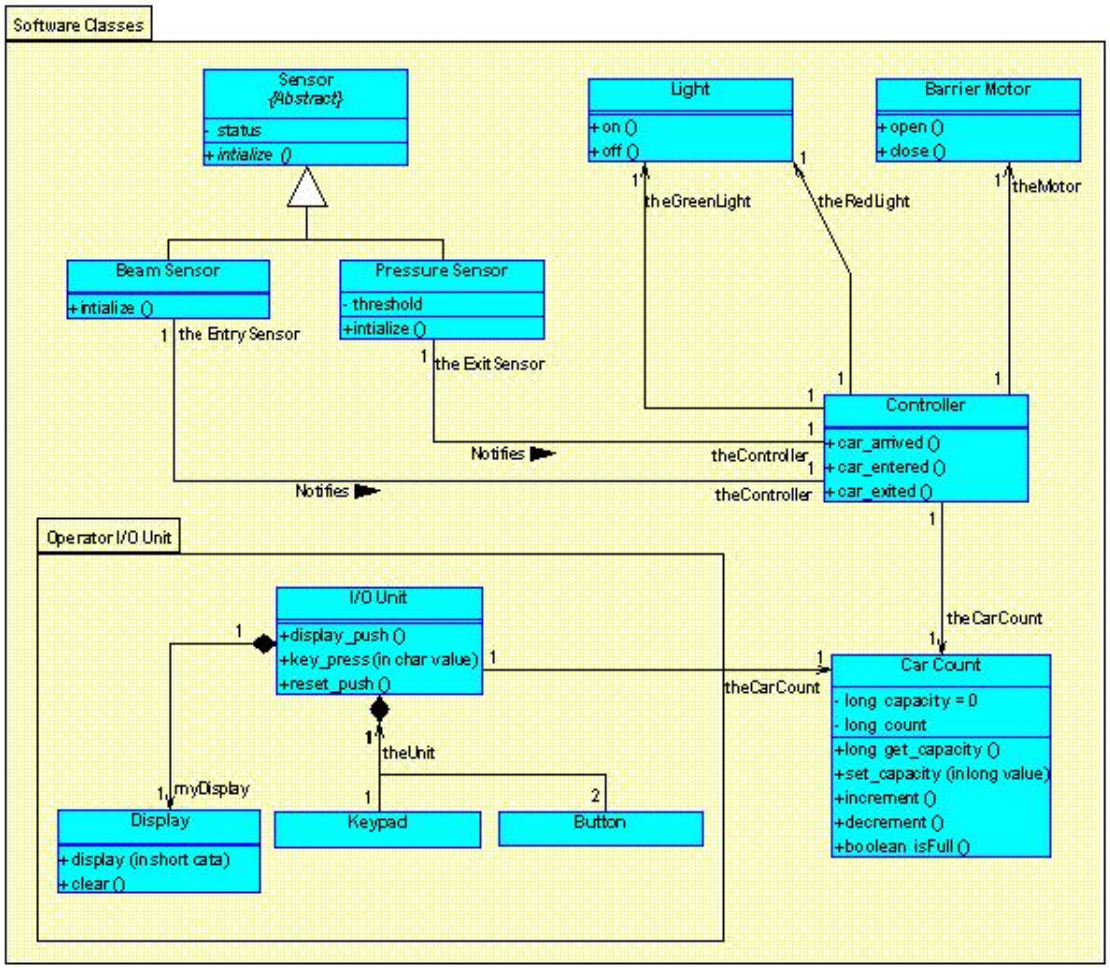
- Identifikation der statischen Struktur der Softwarekomponenten

Methode:

- Identifikation der Objekte
- Festlegung der Attribute der Objekte
- Festlegung der Methoden
- Spezifikation der Interaktion von verschiedenen Objekten
- Spezifikation von Aggregationen von verschiedenen Objekten

Zeitpunkt: Designphase

Class diagram (2)



State diagram

Ziele:

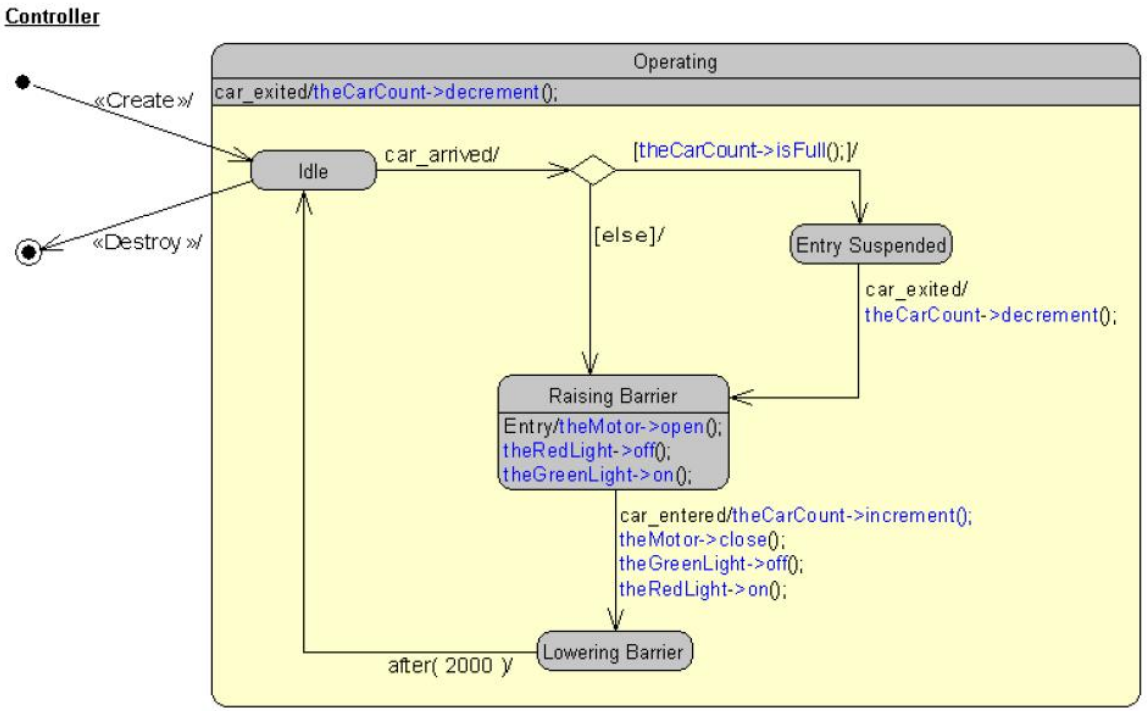
- Identifikation des dynamischen Verhaltens von Objekten

Methoden:

- Identifikation der Zustände einer Komponente
- Identifikation der möglichen Zustandsübergänge
- Identifikation der Ereignisse die zu Übergängen führen

Zeitpunkt: Designphase

State diagram (2)



Concurrency diagram

Ziele:

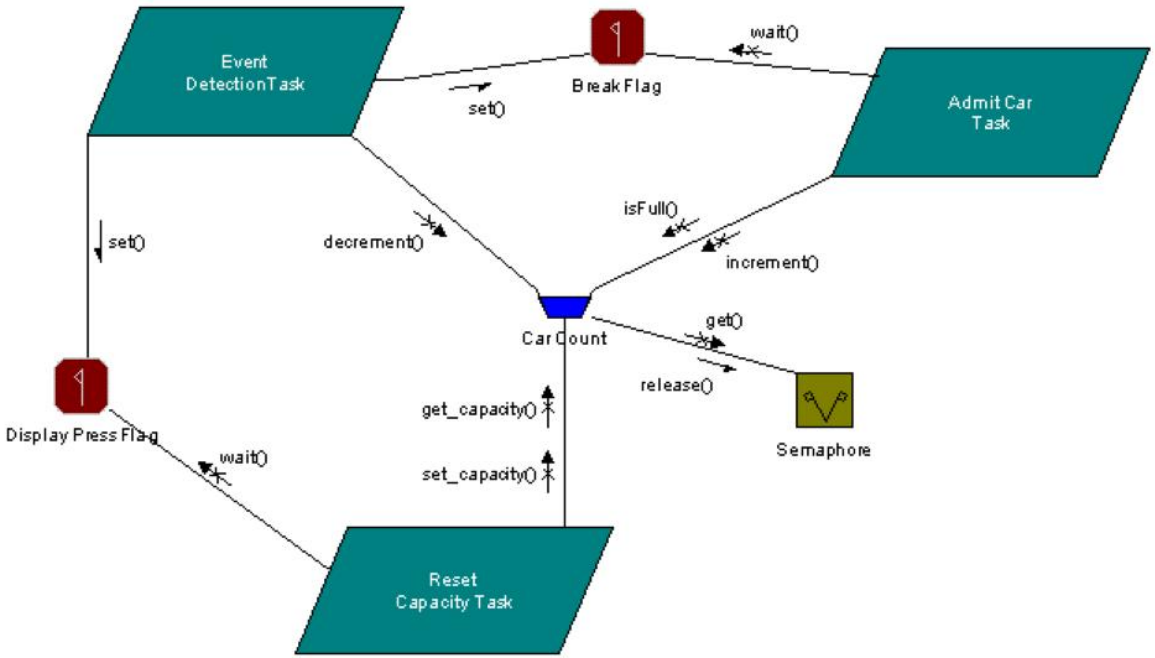
- Spezifikation von Prozessen und deren Interaktion

Methoden:

- Identifikation der einzelnen Prozesse
- Spezifikation der Prozessinterkommunikation
- Spezifikation des wechselseitigen Ausschlusses
- Synchronisation von Prozessen

Zeitpunkt: Designphase

Concurrency diagram (2)



Werkzeuge

- Graphische Darstellung
- Automatische Dokumentengeneration
- Codegeneratoren

Vorlesung Echtzeitsysteme - Nebenläufigkeit

Prof. Alois Knoll

Lehrstuhl Embedded Systems and Robotics

Sommersemester 2005

Inhalt

- Motivation
- Unterbrechungen (Interrupts)
- (Software-)Prozesse
- Threads
- Interprozesskommunikation (IPC)

Literatur

Literatur:

- R.G.Herrtwich, G.Hommel, Kooperation und Konkurrenz,1989
- A.S.Tanenbaum, Moderne Betriebssysteme, 2002

Links:

- http://www.beyondlogic.org/interrupts/interupt.htm
- http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html

Definition von Nebenläufigkeit:

Allgemeine Bedeutung: Nebenläufigkeit bezeichnet das Verhältnis von Ereignissen, die nicht kausal abhängig sind, die sich also nicht beeinflussen. Ereignisse sind nebenläufig, wenn keines eine Ursache des anderen ist. Oder anders ausgedrückt: Aktionen können nebenläufig ausgeführt werden (sie sind parallelisierbar), wenn keine das Resultat der anderen benötigt.

Bedeutung in der Programmierung: Nebenläufigkeit bezeichnet hier die Eigenschaft von Programmcode nicht linear hintereinander ausgeführt zu werden, sondern parallel ausführbar zu sein. Die Nebenläufigkeit von mehreren unabhängigen Prozessen bezeichnet man als **Multitasking**; Nebenläufigkeit innerhalb eines Prozesses als **Multithreading**.

Motivation

Gründe für Nebenläufigkeit in Echtzeitsystemen:

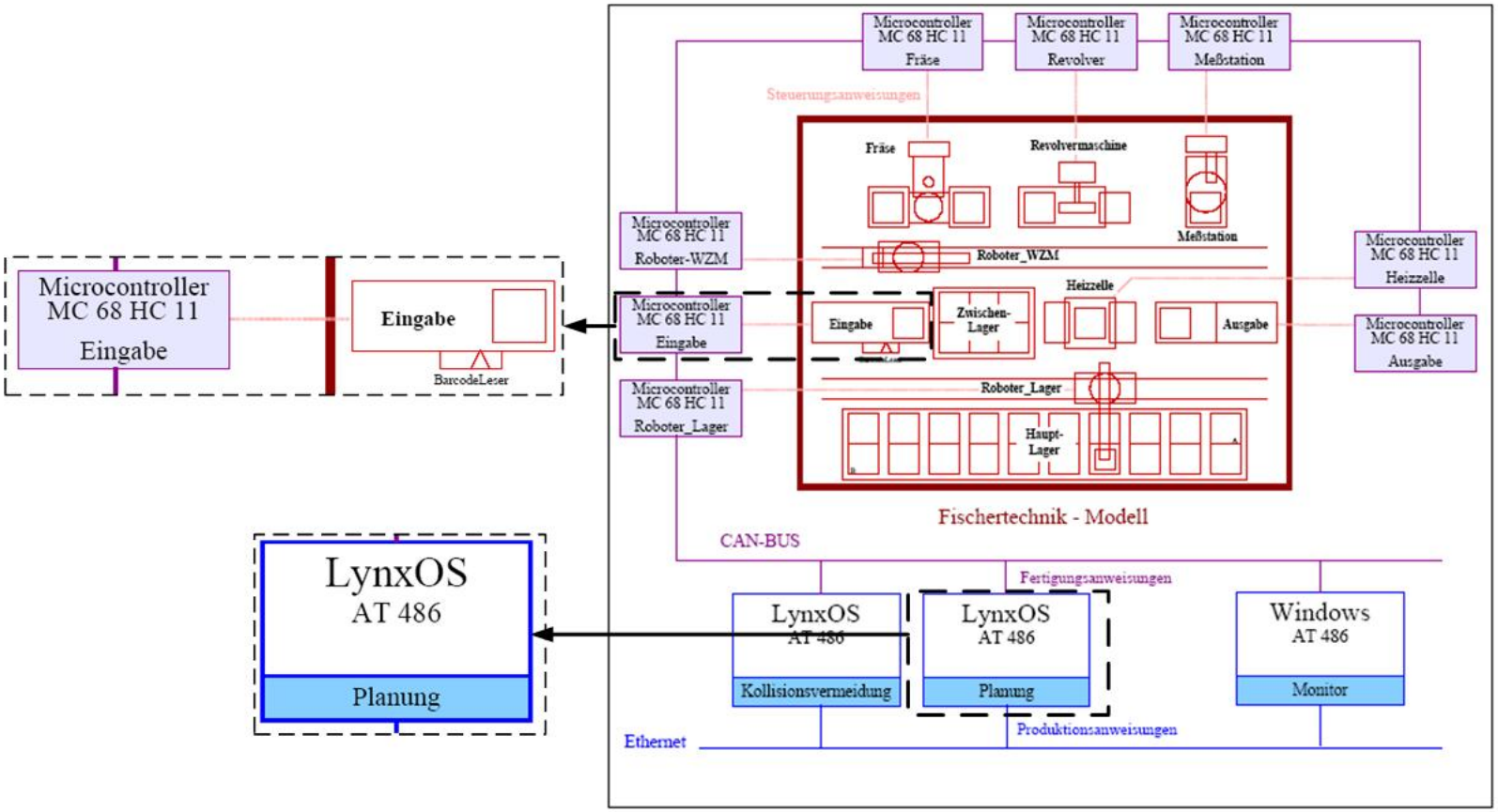
- Echtzeitsysteme sind häufig verteilte Systeme (mehrere Prozessoren).
- Zumeist werden zeitkritische und zeitunkritische Aufgaben parallel berechnet.
- Bei reaktiven Systemen ist die maximale Antwortzeit häufig limitiert.

Anwendungsfälle für Nebenläufigkeit (Unterbrechungen)

Signal falls Temperaturwert überschritten wird
$\Rightarrow$ Unterbrechungen (interrupts)

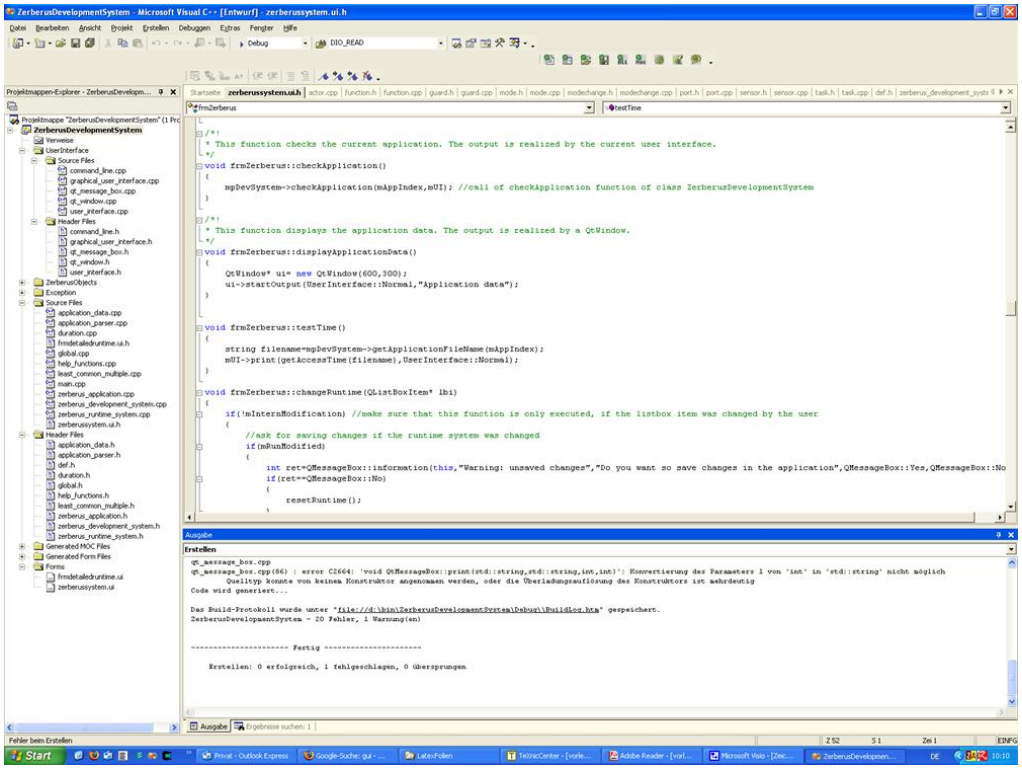
Allgemeines Anwendungsgebiet: hauptsächlich zur Anbindung von externer Hardware

Anwendungsfälle für Nebenläufigkeit (Prozesse)



Verteiltes System zur Steuerung der Industrieanlage ⇒ Prozesse (tasks)
Allgemeine Anwendungsgebiete: verteilte Systeme, unterschiedlichen Anwendungen auf einem Prozessor

Anwendungsfälle für Nebenläufigkeit (Threads)



Reaktion auf Nutzereingaben trotz Berechnungen
$\Rightarrow$ leichtgewichtige Prozesse (Threads)

Allgemeines Anwendungsgebiet: unterschiedliche Berechnungen im gleichen Anwendungskontext

Unterbrechungen (Interrupts)

Anbindung an die Umwelt

Es muss ein Mechanismus gefunden werden, der es erlaubt Änderungen der Umgebung (z.B. einen Mausklick) zu registrieren.

1. Ansatz: **Polling**
Es werden die IO-Register reihum nach Änderungen abgefragt und bei Änderungen spezielle Antwortprogramme ausgeführt.

- Vorteile:
  - bei wenigen IO-Registern sehr kurze Latenzzeiten
  - bei einer unerwarteten Ereignisflut wird das Zeitverhalten des Programms nicht beeinflusst
  - Kommunikation erfolgt synchron mit der Programmausführung
- Nachteile:
  - die meisten Anfragen sind unnötig
  - hohe Prozessorbelastung
  - Reaktionszeit steigt mit der Anzahl an Ereignisquellen

Lösung Interruptkonzept

Interrupt: Ein Interrupt ist ein durch ein Ereignis ausgelöster, automatisch ablaufender Mechanismus, der die Verarbeitung des laufenden Programms unterbricht und die Wichtigkeit des Ereignisses überprüft. Darauf basierend erfolgt die Entscheidung, ob das bisherige Programm weiter bearbeitet wird oder eine andere Aktivität gestartet wird.

Vorteile:

- sehr geringe Extrabelastung der CPU
- Prozessor wird nur dann beansprucht, wenn es nötig ist

Nachteile:

- in der Regel längere Reaktionszeiten

Technische Realisierung

Zur Realisierung besitzen Rechner einen oder mehrere spezielle Interrupt-Eingänge. Wird ein Interrupt aktiviert, so führt dies zur Ausführung der entsprechenden **Unterbrechungsbehandlungs-routine (interrupt handler, interrupt service routine (ISR))**.

Das Auslösen der Unterbrechungsroutine ähnelt einem Unterprogrammaufruf. Der Programmablauf wird an einer anderen Stelle fortgesetzt und nach Beendigung der Routine an der unterbrochenen Stelle fortgefahren. Allerdings tritt die Unterbrechungsroutine im Gegensatz zum Unterprogrammsaufruf asynchron, also an beliebigen Zeitpunkten, auf.

Sperren von Interrupts

Durch die Eigenschaft der Asynchronität kann eine deterministische Ausführung nicht gewährleistet werden. Aus diesem Grund kann eine kurzfristige Sperrung von Interrupts nötig sein, um eine konsistente Ausführung der Programme zu erlauben.

Durch eine Verzögerung werden Interrupts in der Regel nur verzögert, nicht jedoch gelöscht.

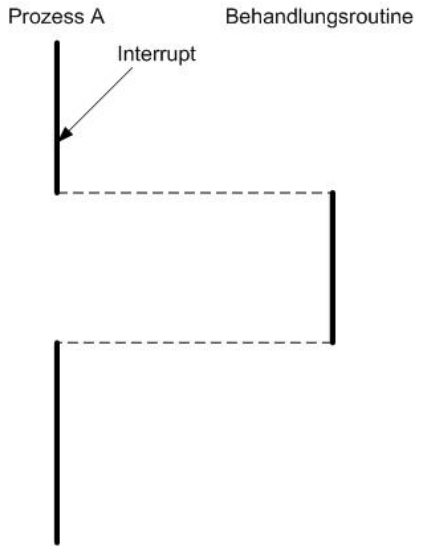
Interrupt Prioritäten

Unterbrechnungen besitzen unterschiedliche Prioritäten. Beim Auftreten einer Unterbrechung werden Unterbrechungen gleicher oder niedrigerer Priorität gesperrt.

Tritt dagegen während der Ausführung der Behandlungsroutine eine erneute Unterbrechung mit höherer Priorität auf, so wird die Unterbrechungsbehandlung gestoppt und die Behandlungsroutine für die Unterbrechung mit höherer Priorität durchgeführt.

Ablauf einer Unterbrechung

1. Sperren von Unterbrechungen mit gleicher oder niedrigerer Priorität
2. Retten des Prozessorstatus
3. Bestimmen der Interruptquelle
4. Laden des Interruptvektors (Herstellung des Anfangszustandes für Behandlungsroutine)
5. Abarbeiten der Routine
6. Rückkehr zur Programmausführung (nicht unbedingt der unterbrochene Prozess)



Hardware Interrupts

Nachfolgend wird eine typische Belegung (Quelle von 2002) der Interrupts angegeben:

00	Systemtaktgeber
01	Tastatur
02	programmierbarer Interrupt-Controller
03	serielle Schnittstelle COM2 (E/A-Bereich 02F8)
04	serielle Schnittstelle COM1 (E/A-Bereich 03F8)
05	frei, oft Soundkarte (Soundblaster-Emulation) oder LPT2
06	Diskettenlaufwerk
07	parallele (Drucker-)Schnittstelle LPT1 (E/A-Bereich 0378)
08	Echtzeitsystemuhr
09	frei
10	frei
11	frei
12	PS/2-Mausanschluss
13	Koprozessor (ob separat oder in CPU integriert)
14	primärer IDE-Kanal
15	sekundärer IDE-Kanal

Programmieren von Interrupts

```
void interrupt yourisr()   /* Interrupt Service Routine (ISR) */
{
  disable();

  /* Body of ISR goes here */

  oldhandler();

  outportb(0x20,0x20);     /* Send EOI to PIC1 */
  enable();
}
```

Erläuterung

- void interrupt yourisr: Deklaration einer Interrupt Service Routine
- disable(): Ist eine weitere Unterbrechung von höher priorisierten Interrupts nicht gewünscht, so können auch diese gesperrt werden.
- oldhandler(): Oftmals benutzen mehrere Programme einen Interrupt (z.B. die Uhr), in diesem Fall sollte man die bisherige ISR sichern (siehe nächste Folie) und an den neuen ISR anhängen
- outportb(): Dem PIC (Programmable Interrupt Controler) muss signalisiert werden, dass die Behandlung des Interrupts beendet ist.
- enable: Die Interrupt-Sperre muss aufgehoben werden.

Einfügen der Routine in Interrupt Vector Table

```
#include <dos.h>
#define INTNO 0x0B    /* Interupt Number 3*/

void main(void)
{
 oldhandler = getvect(INTNO);  /* Save Old Interrupt Vector */
 setvect(INTNO, yourisr);      /* Set New Interrupt Vector Entry */
 outportb(0x21,(inportb(0x21) & 0xF7)); /* Un-Mask (Enable) IRQ3 */

 /* Set Card - Port to Generate Interrupts */
 /* Body of Program Goes Here */
 /* Reset Card - Port as to Stop Generating Interrupts */

 outportb(0x21,(inportb(0x21) | 0x08)); /* Mask (Disable) IRQ3 */

 setvect(INTNO, oldhandler); /*Restore old Vector Before Exit*/
}
```

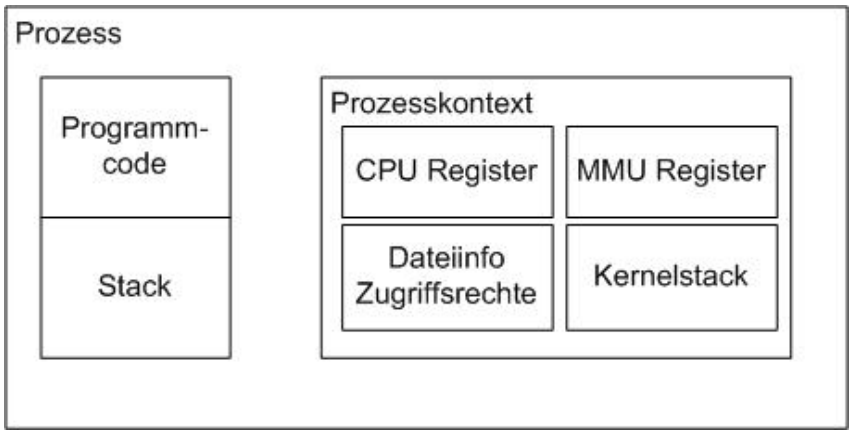
Erläuterung

- INTNO: Es soll der Hardwareinterrupt IRQ 3 (serielle Schnittstelle) verwendet werden, dieser Interrupt entspricht der Nummer 11 (da insgesamt 255 Interrupts (vor allem Softwareinterrupts) vorhanden).
- oldhandler=getvect(INTNO): Durch die Funktion getvect() kann die Adresse der Behandlungsfunktion zurückgelesen werden. Diese wird in der vorher angelegte
- setvect: setzen der neuen Routine
- outportb: setzen einer neuen Maskierung

Prozesse

Prozess: Abstraktion eines sich in Ausführung befindlichen Programms

Die gesamte Zustandsinformation der Betriebsmittel für ein Programm wird als eine Einheit angesehen und als Prozess bezeichnet.
Prozesse können weitere Prozesse erzeugen $\Rightarrow$ Vater-,Kinderprozesse.



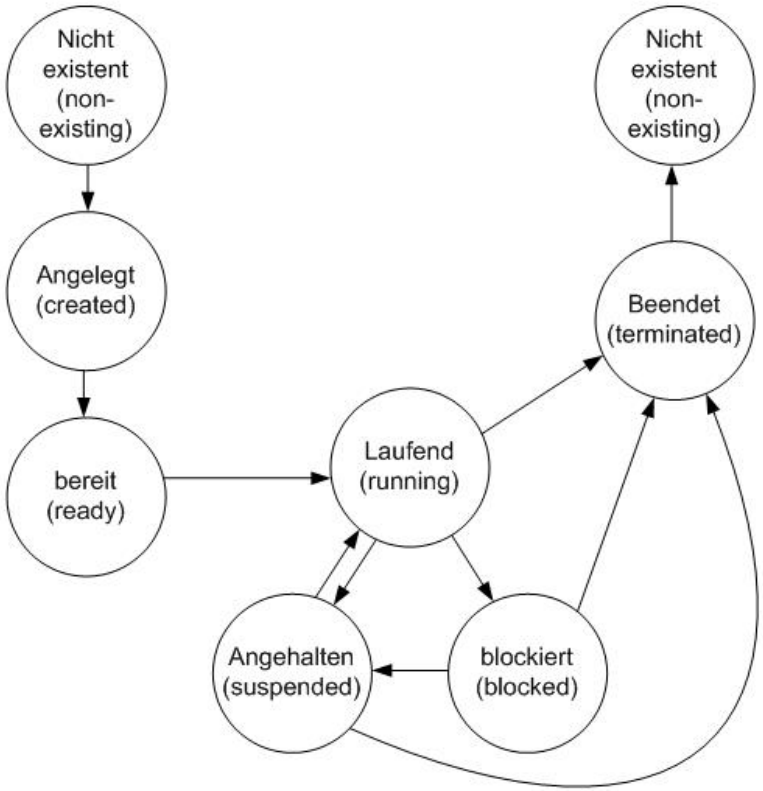
Prozessausführung: Die Prozessausführung benötigt Resourcen:

- Prozessorzeit
- Speicher
- sonstige Betriebsmittel (z.B. spezielle Hardware)

Die **Ausführungszeit** ist neben dem Programm abhängig von:

- Leistungsfähigkeit des Prozessors
- Verfügbarkeit der Betriebsmittel
- Eingabeparametern
- Verzögerungen durch andere (wichtigere) Aufgaben

Prozesszustände



Fragen bei der Implementierung

- Welche Betriebsmittel sind notwendig?
- Welche Ausführungszeiten besitzen einzelne Prozesse?
- Wie können Prozesse kommunizieren?
- Wann soll welcher Prozess ausgeführt werden?
- Wie können Prozesse synchronisiert werden?

Klassifikation von Prozessen

- periodisch vs. aperiodisch
- statisch vs. dynamisch
- Wichtigkeit der Prozesse (kritisch, nötig, nicht nötig)
- parallele vs. nebenläufige Ausführung

Klassifikation der Systeme

Prozesse können auf

- einem Rechner (Pseudoparallelismus)
- einem Multiprozessorsystem mit Zugriff auf gemeinsamen Speicher
- oder auf einem Multiprozessorsystem ohne gemeinsamen Speicher

ausgeführt werden.

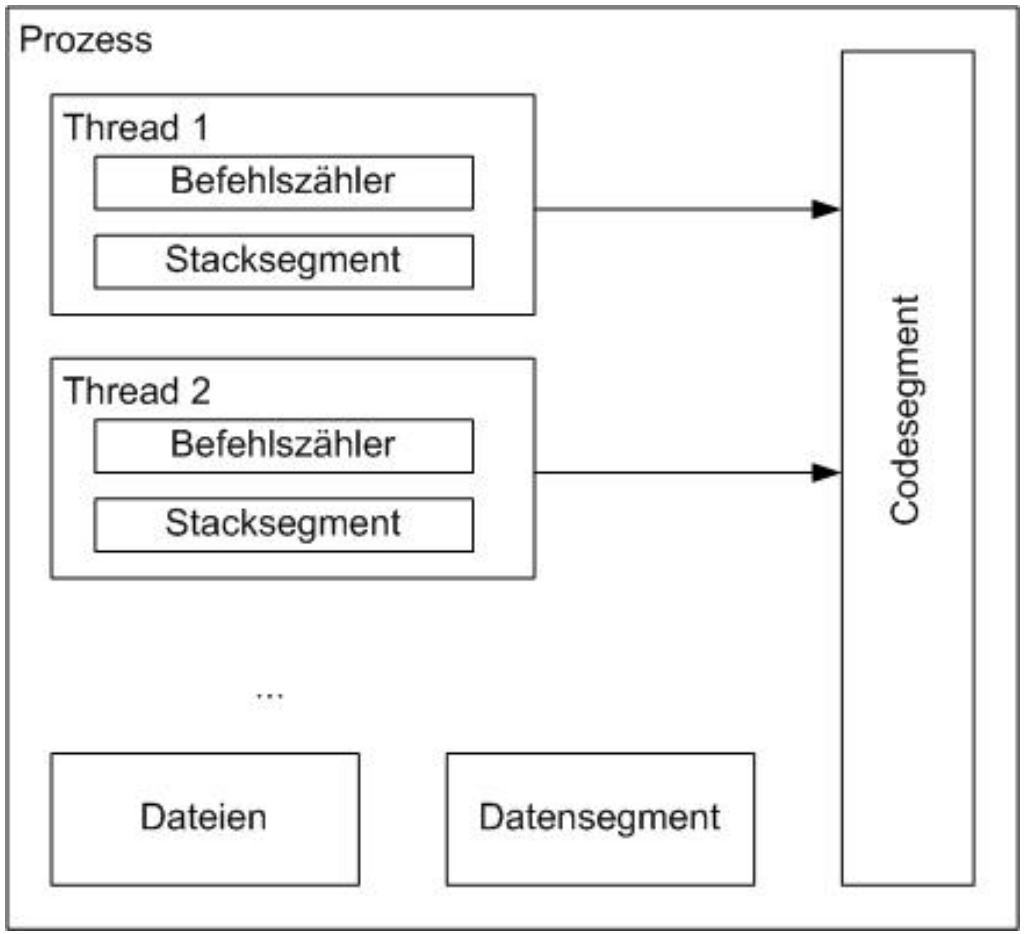
Threads

Leichtgewichtige Prozesse (Threads)

- Der Speicherbedarf von Prozessen ist in der Regel groß (CPU-Daten, Statusinformationen, Angaben zu Dateien und EA-Geräten...).
- Bei Prozesswechsel müssen die Prozessdaten ausgetauscht werden $\Rightarrow$ hohe Systemlast, zeitaufwendig.
- Viele Systeme erfordern keine komplett neuen Prozesse. Vielmehr sind Programmabläufe nötig, die auf den gleichen Prozessdaten arbeiten.

$\Rightarrow$ Einführung von Threads

Threads


Prozess
Thread 1
Befehlszähler
Stacksegment
Thread 2
Befehlszähler
Stacksegment
...
Dateien
Datensegment
Codesegment

Prozesse vs. Threads

- Verwaltungsaufwand von Threads ist deutlich geringer
- Effizienzvorteil: bei einem Threadwechsel ist kein vollständiger Austausch des Prozesskontextes notwendig.
- Kommunikation zwischen Threads des gleichen Prozesses kann über gemeinsamen Speicher erfolgen.
- Zugriffe auf den Speicherbereich anderer Prozesse führen zu Fehlern.
- Probleme bei Threads: durch die gemeinsame Nutzung von Daten kann es zu Konflikten kommen.

Probleme

Race Conditions: Situationen, in denen zwei oder mehrere Prozesse die gleichen geteilten Daten lesen oder schreiben und das Resultat davon abhängt wann genau welcher Prozess ausgeführt wurde, werden Race Condition genannt.

Lösung: Einführung von **kritischen Bereichen** und wechselseitiger Ausschluss.

Bedingungen an Lösung für wechselseitigen Ausschluss

An eine gute Lösung für den wechselseitigen Ausschluss können insgesamt vier Bedingungen gestellt werden:

1. Es dürfen niemals zwei Prozesse gleichzeitig im kritischen Bereich sein.
2. Es dürfen keine Annahmen über die Geschwindigkeit oder Anzahl der Prozessoren gemacht werden.
3. Kein Prozess darf ausserhalb von kritischen Regionen andere Prozesse blockieren.
4. Kein Prozess soll unendlich auf das Eintreten in den kritischen Bereich warten müssen.

Lösung für wechselseitigen Ausschluss

1. Ausschalten von Interrupts
2. Semaphore/TSL (Test Set Lock)
3. Monitore

Ausschalten von Interrupts zum wechselseitigen Ausschluss

Die einfachste Möglichkeit einen Kontextwechsel zu verhindern ist das Ausschalten von Interrupts.

Vorteile:

- einfach zu implementieren, keine weiteren Konzepte sind nötig
- schnelle Ausführung

Nachteile:

- Für Multiprozessorsysteme ungeeignet
- Keine Gerätebehandlung während der Sperre
- Lange Sperren kritisch bei Echtzeitanwendungen

Semaphore

- Von Edsger W. Dijkstra 1965 eingeführt.
- Ein Semaphore ist eine Datenstruktur bestehend aus einem Zähler und den Operationen P(prolaag, erniedrigen) und V (verhoog, erhöhen).
- Die Operationen P und V müssen atomar ausführbar sein (die Ausführung darf nicht unterbrochen werden).
- Mit dem Semaphore assoziierte Warteschlangen ermöglichen eine Realisierung ohne Busy-Waiting

```
Semaphore S
{
  int z;

  Semaphore init(int n)
  {
    z=n;
  }

  P()
  {
    while(z==0); /*wait until z>0*/
    z=z-1;
  }

  V()
  {
    z=z+1;
  }
}
```

Lösung des WA mit Semaphoren

Der kritische Bereich kann durch den Einsatz von Semaphoren geschützt werden.
Dazu sind folgende Schritte nötig:

1. Jedem kritischen Bereich wird ein Semaphor zugewiesen.
2. Vor dem Eintreten in den kritischen Bereich muss dieser Semaphore angefordert werden.
3. Nach dem Verlassen des kritischen Bereiches muss der Semaphore wieder freigegeben werden.

$\Rightarrow$ Ein paralleler Zugriff auf den kritischen Bereich wird durch den Einsatz von binären Semaphoren verhindert.

Monitore

Problem mit Semaphoren: durch unvorsichtige Implementierungen können leicht Deadlocks entstehen

$\Rightarrow$ Einführung des Highlevelkonzeptes Monitor

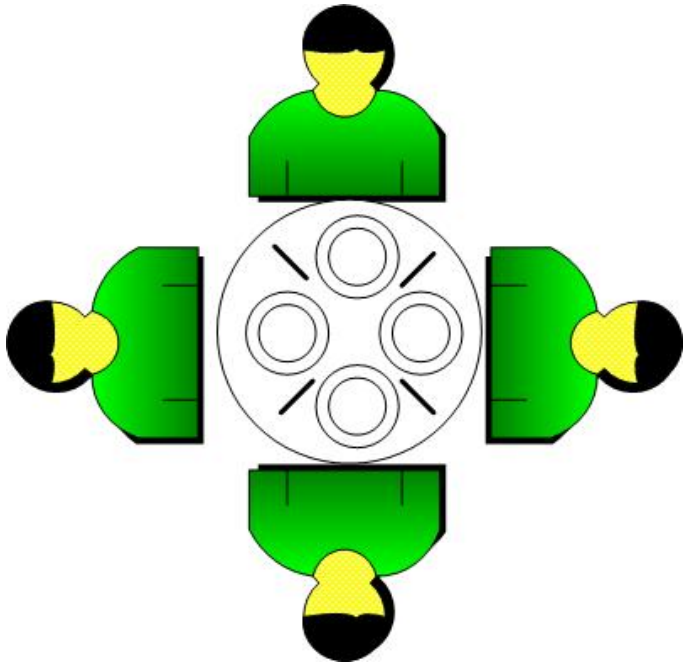
Ein **Monitor** ist eine Menge von Prozeduren und Variablenstrukturen, wobei in jedem Moment nur jeweils ein Prozess aktiv den Monitor betreten kann.

Typischerweise realisiert der Compiler den Monitor automatisch durch die Verwendung von binären Semaphoren.

Weitere Probleme

- Deadlocks (Verklemmung): Situation in der mindestens zwei Prozesse auf ein Betriebsmittel warten, dass ein anderer beteiligter Prozess belegt hat.
- Starvation (Aussperrung): Situtation in der ein Prozess unendlich lange auf ein Betriebsmittel wartet.
- Priority Inversion: siehe Vorlesung Scheduling

Klassisches Beispiel: Speisende Philosophen



Die Philosophen sitzen an einem Tisch und diskutieren. Jedes Mal wenn ein Philosoph hungrig ist greift er zuerst nach dem linken, dann nach dem rechten Stäbchen und fängt an zu essen.

Deadlock: Greifen alle Philosophen gleichzeitig nach dem linken Stäbchen so verhungern sie.

Starvation: Behält ein Philosoph ein Stäbchen, so verhungert der entsprechende Nachbar.

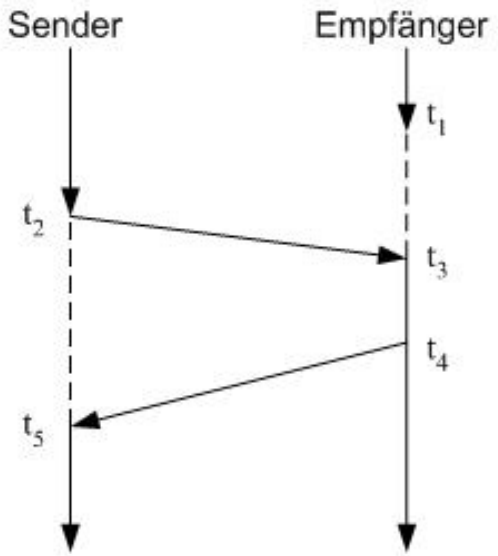
Interprozesskommunikation (IPC)

Klassifikation der Kommunikation

- synchrone vs. asynchrone Kommunikation
- pure Ereignisse vs. wertbehaftete Nachrichten

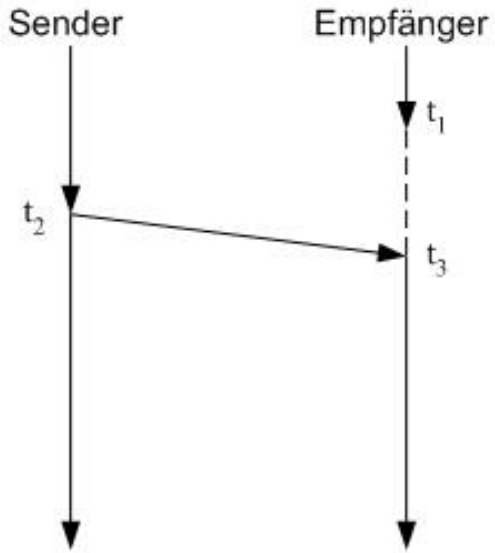
Synchron vs. Asynchron

Synchrone Kommunikation



t_1 : Empfänger wartet auf Nachricht
t_2 : Sender schicht Nachricht und blockiert
t_3 : Empfänger bekommt Nachricht, die Verarbeitung startet
t_4 : Verarbeitung beendet, Antwort wird gesendet
t_5 : Sender empfängt Nachricht und arbeitet weiter

Asynchrone Kommunikation



t_1 : Empfänger wartet auf Nachricht
t_2 : Sender schicht Nachricht und arbeitet weiter
t_3 : Empfänger bekommt Nachricht, die Verarbeitung startet

IPC-Mechanismen

- Datenströme: direkter Datenaustausch, Pipes, MessageQueues
- Ereignisse: Signale, Semaphore
- Funktionsaufrufe: RPC, Corba

Kommunikation durch Datenströme

Direkter Datenaustausch

Mit Semaphoren und Monitoren geschützte Datenstrukturen eignen sich sehr gut für den Austausch von Daten:

- schnelle Kommunikation da auf den Speicher direkt zugegriffen werden kann.

Allerdings kann die Kommunikation nur lokal erfolgen und zudem müssen die Prozesse eng miteinander verknüpft sein.

Als Lösung bietet sich der Nachrichtenaustausch an. Grundsätzlich erfolgt der Austausch über das Ausführen von Funktionen send(receiver_address, &message) und receive(sender_address, &message).

Fragestellungen beim Nachrichtenaustausch

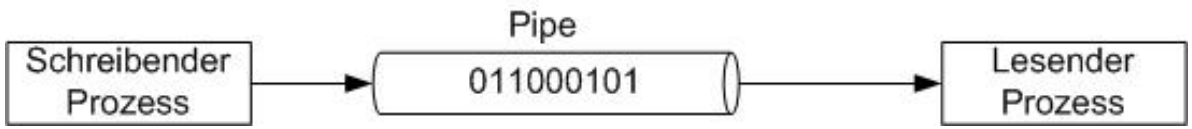
- Lokale oder verteilte Kommunikation?
- Festlegung des Protokolls:
  - mit/ohne Bestätigung
  - Nachrichtenverluste
  - Zeitintervalle
  - Reihenfolge der Nachrichten
- Adressierung
- Authentifikation
- Performance
- Sicherheit (Verschlüsselung)

Heute: vor allem lokale Kommunikation, echtzeitfähige Protokolle zur Kommunikation in eigener Vorlesung

Pipes

Die **Pipe** bezeichnet eine gepufferte, unidirektionale Datenverbindung zwischen zwei Prozessen nach dem First-In-First-Out- (FIFO-)Prinzip.

Über den Namen der Pipe (ähnlich einem Dateinamen) können Prozesse unterschiedlichen Ursprungs auf eine Pipe lesend oder schreibend zugreifen. Zur Kommunikation zwischen Prozessen gleichen Ursprungs (z.B. Vater-,Kindprozess) können auch anonyme Pipes verwendet werden. Die Kommunikation erfolgt immer asynchron.



Pipes in Posix

POSIX (Portable Operating System Interface) versucht durch Standardisierung der Systemaufrufe die Portierung von Programmen zwischen verschiedenen Betriebssystemen zu erleichtern.

POSIX.1 definiert folgende Funktionen für Pipes:

```
int mkfifo(char* name, int mode);  /*Erzeugen einer benannten Pipe*/
int unlink ( char *name );         /*Loeschen einer benannten Pipe*/
int open ( char *name, int flags); /*Oeffnen einer benannten Pipe*/
int close ( int fd );              /*Schliessen des Lese- oder
                                   Schreibendes einer Pipe*/
int read ( int fd, char *outbuf,
     unsigned bytes );             /*Lesen von einer Pipe*/
int write ( int fd, char *outbuf,
     unsigned bytes );             /*Schreiben an eine Pipe*/
int pipe ( int fd[2] );            /*Erzeugen eine unbenannte Pipe*/
```

Nachteile von Pipes

Pipes bringen einige Nachteile mit sich:

- Es können keine Daten aufgehoben werden.
- Beim Öffnen blockiert der Funktionsaufruf, bis auch das zweite Zugriffsende geöffnet wird (Verhinderung durch O_NDELAY Flag).
- Pipes sind nicht nachrichtenorientiert.
- Daten sind nicht priorisierbar.

Lösung: Nachrichtenwartschlangen

Nachrichtenwarteschlangen (Message Queues)

Nachrichtenschlangen (Message Queues) sind eine Erweiterung von Pipes. Im Folgenden werden Nachrichtenwarteschlangen, wie in POSIX 1003.1b (Echtzeiterweiterung von POSIX) definiert, betrachtet.

Eigenschaften der POSIX MessageQueues:

- Beim Anlegen einer MessageQueue wird der benötigte Speicher reserviert. $\Rightarrow$ Speicher muss nicht erst beim Schreibzugriff angelegt werden.
- Die Kommunikation erfolgt nachrichtenorientiert. Die Anzahl der vorhandenen Nachrichten kann dadurch abgefragt werden.
- Nachrichten sind priorisierbar. $\Rightarrow$ Es können leichter Zeitgarantien gegeben werden.

Message Queues in POSIX

POSIX definiert folgende Funktionen für Nachrichtenwartschlangen:

```
mqd_t mq_open(const char *name,
   int oflag, ...);                                  /*Oeffnen einer Message Queue*/
int mq_close(mqd_t mqdes);                           /*Schliessen einer Message
                                                       Queue*/
int mq_unlink(const char *name);                     /*Loeschen einer Nachrichten-
                                                       warteschlange*/
int mq_send(mqd_t mqdes, const char *msg_ptr,
   size_t msg_len, unsigned int msg_prio);           /*Senden einer Nachricht*/
size_t mq_receive(mqd_t mqdes, char *msg_ptr,
   size_t msg_len, unsigned int *msg_prio);          /*Empfangen einer Nachricht*/
int mq_setattr(mqd_t mqdes, const struct
   mq_attr *mqstat, struct mq_attr *mqstat);         /*Aendern der Attribute*/
int mq_getattr(mqd_t mqdes,
   struct mq_attr *mqstat);                          /*Abrufen der aktuellen
                                                       Eigenschaften*/
int mq_notify(mqd_t mqdes,
   const struct sigevent *notification);             /*Anforderung eines Signals
                                                       bei Nachrichtenankunft*/
```

Kommunikation durch Ereignisse

Signale

Signale werden in Betriebssystemen typischerweise zur Signalisierung von Ereignissen an Prozessen verwendet.

Signale können verschiedene Ursachen haben:

- Ausnahmen, z.B. Division durch Null (SIGFPE) oder ein Speicherzugriffsfehler (SIGSEGV)
- Reaktion auf Benutzereingaben (z.B. Ctrl / C)
- Signal von anderem Prozess zur Kommunikation
- Signalisierung von Ereignissen durch das Betriebssystem, z.B. Ablauf einer Uhr, Beendigung einer asynchronen IO-Funktion, Nachrichtankunft an leerer Nachrichtenwarteschlange (siehe mq_notify())

Prozessreaktionen auf Signale

Der Prozess hat drei Möglichkeiten auf Signale zu reagieren:

1. Ignorierung der Signale
2. Ausführen einer Signalbehandlungsfunktion
3. Verzögerung des Signals, bis Prozess bereit für Reaktion ist

Zudem besteht die Möglichkeit mit der Standardreaktion auf das bestimmte Signal zu reagieren. Da aber typischerweise die Reaktion auf Signale die Beendigung des Empfängerprozesses ist, sollte ein Programm über eine vernünftige Signalbehandlung verfügen, sobald ein Auftreten von Signalen wahrscheinlich wird.

POSIX Funktionen für Signale

POSIX 1003.1 definiert folgende Funktionen:

Funktion	**Bedeutung**
kill	Senden eines Signals an einen Prozess oder eine Prozessgruppe
sigaction	Spezifikation der Funktion zur Behandlung eines Signals
sigaddset	Hinzufügen eines Signals zu einer Signalmenge
sigdelset	Entfernen eines Signals von einer Signalmenge
sigemptyset	Initialisierung einer leeren Signalmenge
sigfillset	Initialisierung einer kompletten Signalmenge
sigismember	Test ob ein Signal in einer Menge enthalten ist
sigpending	Rückgabe der aktuell angekommenen, aber verzögerten Signale
sigprocmask	Setzen der Menge der vom Prozess blockierten Signale
sigsuspend	Änderung der Liste der blockierten Signale und Warten auf Ankunft und Behandlung eines Signals

Einschränkungen der Standard-Signale

POSIX 1003.1 Signale haben folgende Einschränkungen:

- Es existieren zu wenige Benutzersignale (SIGUSR1 und SIGUSR2)
- Signale besitzen keine Prioritäten
- Blockierte Signale können verloren gehen (beim Auftreten mehrerer gleicher Signale)
- Das Signal enthält keinerlei Informationen zur Unterscheidung von anderen Signalen gleichen Typs (z.B. Absender)

Erweiterungen in POSIX 1003.1b

Zur Benutzung von Echtzeitsystemen sind in POSIX 1003.1b folgende Erweiterungen vorgenommen worden:

- Einen Menge von nach Priorität geordneten Signalen, die Benutzern zur Verfügung stehen (Bereich von SIGRTMIN bis SIGRTMAX)
- Einen Warteschlangenmechanismus zum Schutz vor Signalverlust
- Mechanismen zur Übertragung von weiteren Informationen
- schnellere Signallieferung beim Ablauf eines Timers, bei Ankunft einer Nachricht an einer leeren Nachrichtenwarteschlange, bei Beendigung einer IO-Operation
- Funktionen, die eine schnellere Reaktion auf Signale erlauben

POSIX Funktionen für Signale

POSIX 1003.1b definiert folgende zusätzliche Funktionen:

Funktion	**Bedeutung**
sigqueue	Sendet ein Signal inklusive identifizierende Botschaften an Prozess
sigtimedwait	Wartet auf ein Signal für eine bestimmte Zeitdauer, wird ein Signal empfangen, so wird es mitsamt der Signalinformation zurückgeliefert
sigwaitinfo	Wartet auf ein Signal und liefert das Signal mitsamt Information zurück

Beispiel: Programmierung von Signalen

Im Folgenden wird der Code für ein einfaches Beispiel dargestellt: die periodische Ausführung einer Funktion.

Dazu werden folgende Funktionen implementiert:

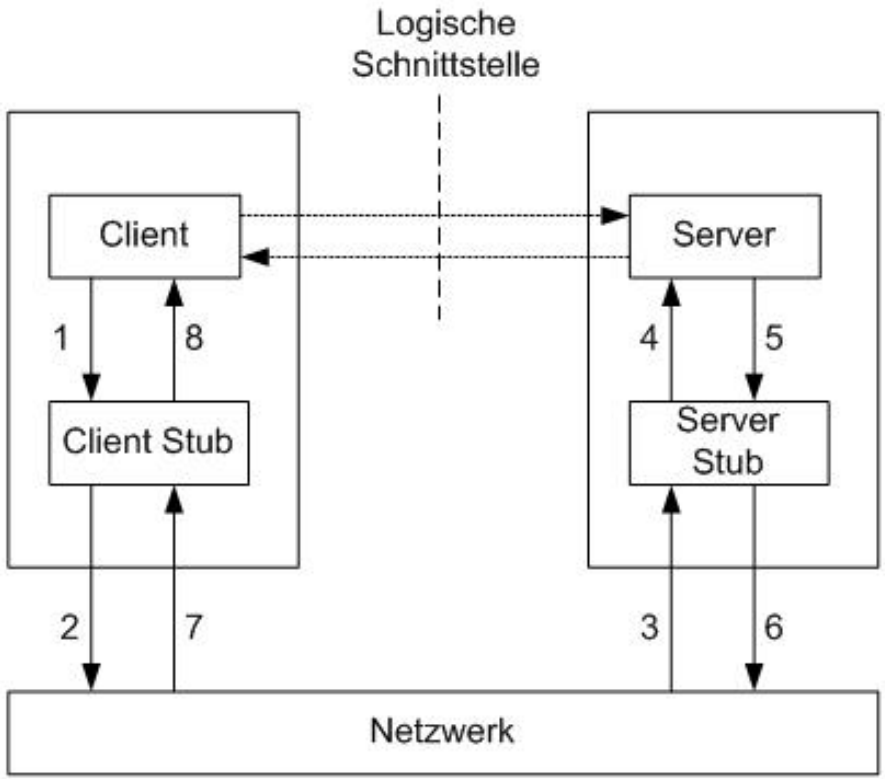
- Initialisierung eines Timers und der Signale
- Setzen eines periodischen Timers
- Warten auf den Ablauf des Timers
- Löschen des Timers
- Hauptfunktion

Semaphore zur Vermittlung von Ereignissen

- Semaphore können neben der Anwendung des wechselseitigen Ausschlusses auch zur Signalisierung von Ereignissen verwendet werden.
- Es ist zulässig, dass Prozesse (Erzeuger) Semaphore andauernd freigeben und andere Prozesse (Verbraucher) Semaphore dauern konsumieren.
- Es können auch benannte Semaphoren erzeugt werden, die dann über Prozessgrenzen hinweg verwendet werden können. Notwendige Funktionen sind dann:
  - sem_open(): zum Erzeugen und / oder Öffnen eines benannten Semaphors
  - sem_unlink(): zum Löschen eines benannten Semaphors

Funktionsaufrufe als Kommunikation

Remote Procedure Call (RPC)



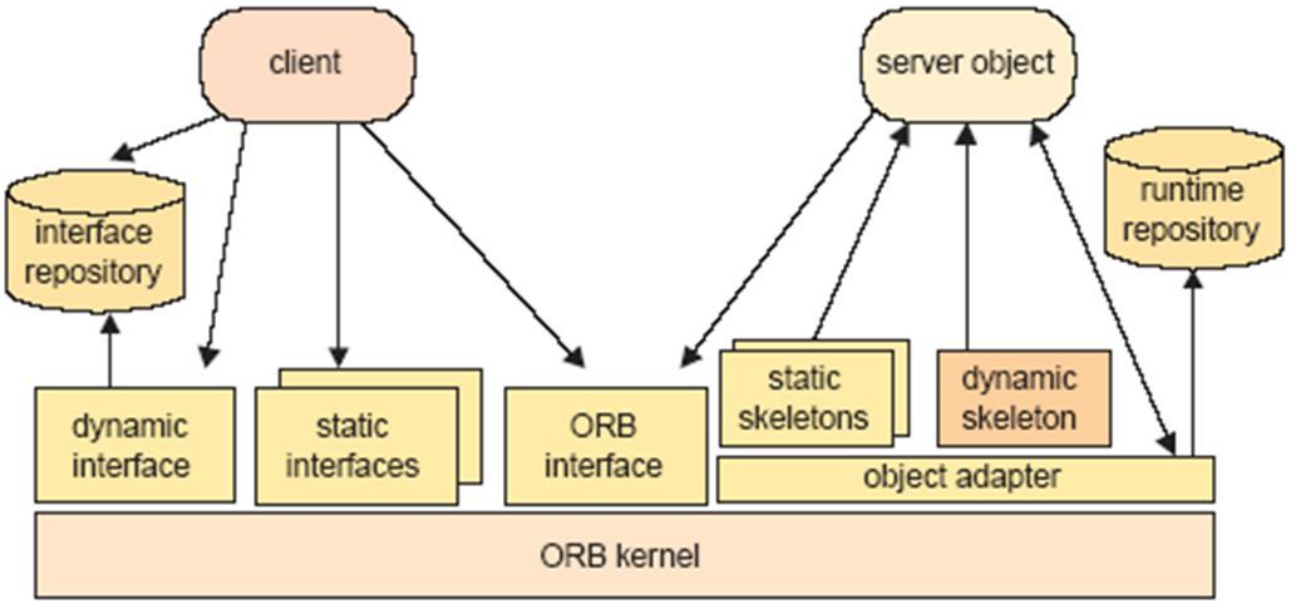
Ablauf RPC

Bei einem Funktionsaufruf über RPC werden folgende Schritte ausgeführt:

1. Lokaler Funktionsaufruf vom Client an Client Stub
2. Konvertierung des Funktionsaufrufs in Übertragungsformat und Senden der Nachricht
3. Empfang der Nachricht von Kommunikationschicht
4. Entpacken der Nachricht und lokaler Funktionsaufruf
5. Übermittlung des Ergebnisses von Server an Server Stub
6. Konvertierung des Funktionsergebnisses in Übertragungsformat und Senden der Nachricht
7. Empfang der Nachricht von Kommunikationschicht
8. Entpacken der Nachricht und Übermittlung des Ergebnisses an Client

Voraussetzung für Echtzeitfähigkeit: Echtzeitfähiges Kommunikationsprotokoll und Mechanismus zum Umgang mit Nachrichtenverlust

Corba (Common Object Request Broker Architecture)



Komponenten in Corba

- **ORB (Object Request Broker)**: vermittelt Anfragen zwischen Server und Client,managt die Übertragung, mittlerweile sind auch echtzeitfähige ORBs verfügbar
- **ORB Interface**: Schnittstelle für Systemdienstaufrufe
- **Interface repository**: speichert die Signaturen der zur Verfügung stehenden Schnittstellen, die Schnittstellen werden dabei in der IDL-Notation (Interface Definition Language) gespeichert.
- **Object Adapter**: Überbrückt die Lücke zwischen Corbaobjekten mit IDL-Schnittstelle und Serverobjekten in der jeweiligen Programmiersprache
- **Runtime repository**: enthält die verfügbaren Dienste und die bereit instantiierten Objekte mitsamt den entsprechenden IDs
- **Skeletons**: enthalten die Stubs für die Serverobjektaufrufe

Zusammenfassung

Zusammenfassung

Folgende Fragen wurden in dieser Vorlesung erklärt und sollten nun verstanden sein:

- Was ist Nebenläufigkeit?
- Welche verschiedenen Arten von Nebenläufigkeit gibt es (+allgemeine Erläuterung)?
- In welchen Fällen wird welche Art verwendet?
- Wie können race conditions vermieden werden?
- Welche Arten der Interprozesskommunikation gibt es(+allgemeine Erklärung)?

Vorlesung Echtzeitsysteme - Scheduling

Prof. Alois Knoll

Lehrstuhl Embedded Systems and Robotics

Sommersemester 2005

Inhalt

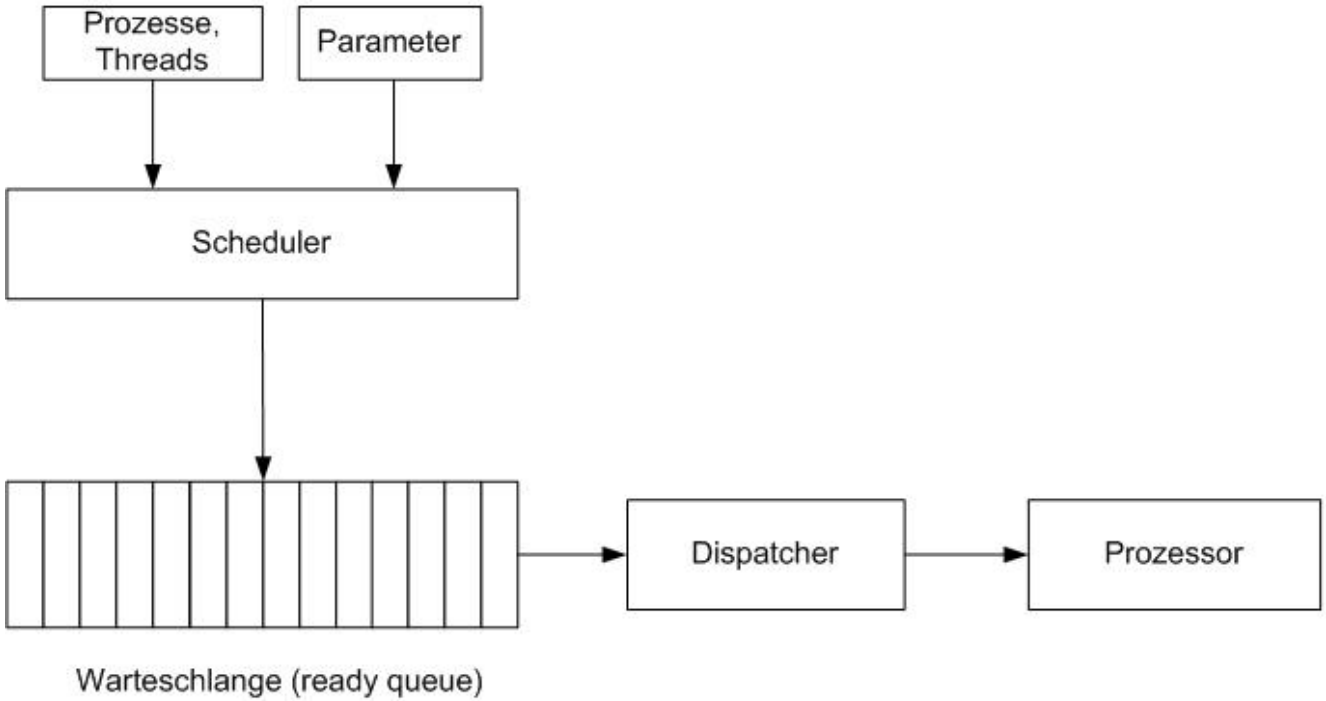
- Definitionen
- Kriterien
- Lösungsstrategien
- Exkurs: Worst Case Execution Times

Literatur

- Jane W. S. Liu, Real-Time Systems, 2000
- Puschner, Peter; Burns, Alan: A review of Worst-Case Execution-Time Analysis, Journal of Real-Time Systems 18 (2000), S.115-128
- Hofmann, Fridolin: Betriebssysteme - Grundkonzepte und Modellvorstellungen, 1991

Definitionen

Scheduler und Dispatcher



Scheduler und Dispatcher

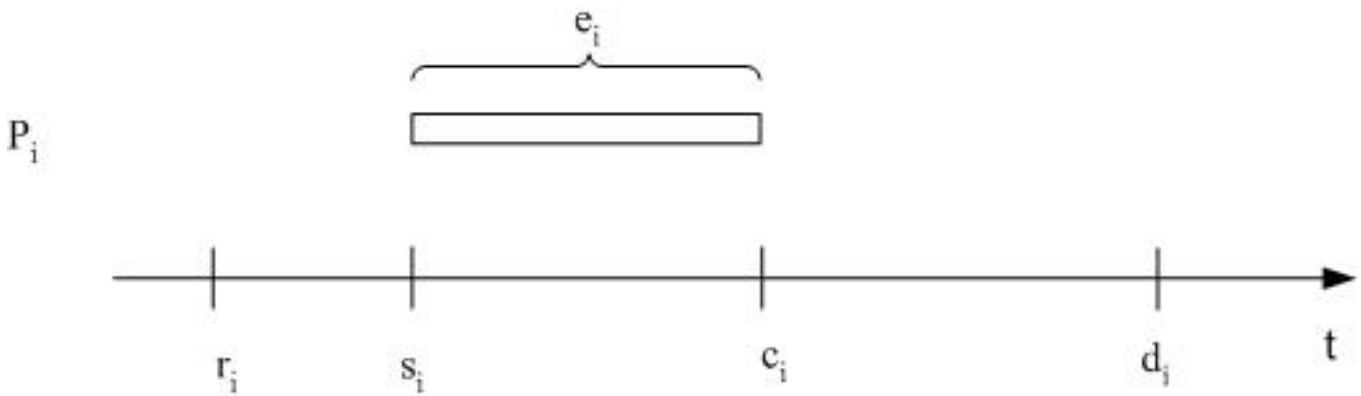
Scheduler: Modul eines Betriebssystems, das die Rechenzeit an die unterschiedlichen Prozesse verteilt. Der ausgeführte Algorithmus wird als **Scheduling-Algorithmus** bezeichnet. Aufgabe des Schedulers ist also die langfristige Planung (Vergleich: Erstellung eines Zugfahrplans).

Dispatcher: Übersetzung: Einsatzleiter, Koordinator, Zuteiler (v.a. im Bereich der Bahn gebräuchlich). Im Rahmen der Prozessverwaltung eines Betriebssystems dient der Dispatcher dazu, bei einem Prozesswechsel dem derzeit aktiven Prozess die CPU zu entziehen und anschließend dem nächsten Prozess die CPU zuzuteilen. Die Entscheidung, welcher Prozess der nächste ist, wird vom Scheduler im Rahmen der Warteschlangenorganisation getroffen.

Zeitliche Bedingungen

Folgende Größen sind charakteristisch für die Ausführung von Prozessen:

1. P_i bezeichnet den i.Prozess (bzw. Thread)
2. r_i: Bereitzeit (ready time) des Prozesses P_i und damit der früheste Zeitpunkt an dem der Prozess dem Prozessor zugeteilt werden kann.
3. s_i: Startzeit: der Prozessor beginnt P_i auszuführen.
4. e_i: Ausführungszeit (execution time): Zeit die der Prozess P_i zur reinen Ausführung auf dem Prozessor benötigt.
5. c_i: Abschlußzeit (completion time): Zeitpunkt zu dem die Ausführung des Prozesses P_i beendet wird.
6. d_i: Frist (deadline): Zeitpunkt zu dem die Ausführung des Prozesses P_i in jeden Fall beendet sein muss.



Mit dem **Spielraum (slack time)** sl_i eines Prozesses P_i wird Zeitraum bezeichnet, um den ein Prozess noch maximal verzögert werden darf:
Die Differenz zwischen der verbleibenden Zeit bis zum Ablauf der Frist und der noch benötigten Ausführungszeit zur Beendigung des Prozesses P_i.
Der Spielraum eines Prozesses, der aktuell durch den Prozessor ausgeführt wird, bleibt konstant, während sich die Spielräume aller nicht-ausgeführten Prozesse verringern.

Kriterien bei der Planung

Faktoren bei der Planung

Für die Planung des Schedulings müssen folgende Faktoren berücksichtigt werden:

- Art der Prozesse (periodisch, nicht periodisch, sporadisch)
- Gemeinsame Nutzung von Ressourcen
- Fristen
- Vorrangrelationen (Präzedenzen: Prozess P_i muss vor P_j ausgeführt werden)

Arten der Planung

Es kann zwischen unterschiedlichen Arten zum Planen unterschieden werden:

- offline vs. online Planung
- statische vs. dynamische Planung
- präemptives vs. nicht-präemptives Scheduling

Offline Planung

Mit der statischen Planung wird die Erstellung eines Ausführungsplanes zur Übersetzungszeit bezeichnet. Zur Ausführungszeit arbeitet der Dispatcher den Ausführungsplan dann ab.

Bereitzeiten, Ausführungszeiten und Abhängigkeit der einzelnen Prozesse müssen schon im voraus bekannt sein.

Die Suche nach einem Schedulingplan ist im Allgemeinen ein NP-hartes Problem. Es werden jedoch keine optimalen Pläne gesucht, vielmehr ist ein gute Lösung (Einhaltung aller Fristen) ausreichend.

Online Scheduling

Alle Schedulingentscheidungen werden online, d.h. auf der Basis der Menge der aktuell lauffähigen Prozess getroffen.

Im Gegensatz zur offline Planung muss wechselseitiger Ausschluss nun über den expliziten Ausschluss (z.B. Semaphoren) erfolgen.

Vorteile:

- Flexibilität
- Bessere Auslastung der Ressourcen

Nachteile:

- Es müssen zur Laufzeit Berechungen zum Scheduling durchgeführt werden $\Rightarrow$ Rechenzeit geht verloren.
- Garantien zur Einhaltung von Fristen sind schwieriger zu geben.

Statische vs. dynamische Planung

Bei der **statischen** Planung basieren alle Entscheidungen auf Parametern, die vor der Laufzeit festgelegt werden.

Bei der **dynamischen** Planung können sich diese Parameter zur Laufzeit ändern.

Präemption

Präemptives (bevorrechtigt, entziehend) Scheduling: Bei jedem Auftreten eines relevanten Ereignisses wird die aktuelle Ausführung eines Prozesses unterbrochen und eine neue Schedulingentscheidung getroffen. Präemptives (unterbrechbares) Abarbeiten:

- Aktionen (Prozesse) haben Prioritäten.
- Prioritäten sind statisch oder werden dynamisch berechnet.
- Ausführung einer Aktion wird sofort unterbrochen, sobald Aktion mit höherer Priorität eintrifft.
- Die unterbrochene Aktion wird an der Unterbrechungsstelle fortgesetzt, sobald keine Aktion höherer Priorität ansteht.
- Typisch für Echtzeitaufgaben (mit Ausnahme von Programmteilen, die zur Sicherung der Datenkonsistenz nicht unterbrochen werden dürfen).
- Nachteil: häufiges Umschalten reduziert Leistung.

Keine Beendigung möglich

Nicht präemptives Scheduling: Ein Prozess, der den Prozessor zugewiesen bekommt, wird solange ausgeführt, bis der Prozess beendet wird oder er aber den Prozess freigibt.
Schedulingentscheidungen werden nur nach der Prozessbeendigung oder dem Übergang des ausgeführten Prozesses in den blockierten Zustand vorgenommen.

- Eine begonnene Aktion wird beendet, selbst wenn während der Ausführung Aktionen höherer Dringlichkeit eintreffen
- z.B. Schreiben in Druckerpuffer, Schließen eines Ventils
- Nachteil: evtl. Versagen (zu lange Reaktionszeit) des Systems beim Eintreffen unvorhergesehener Anforderungen

Schedulingkriterien in Nicht-Echtzeitsystemen

- Fairness: gerechte Verteilung der Prozessorzeit
- Effizienz: vollständige Auslastung der CPU
- Antwortzeit: interaktive Prozesse sollen schnell reagieren
- Verweilzeit: Aufgaben im Batchbetrieb sollen möglichst schnell ein Ergebnis liefern
- Durchsatz: Maximierung der Anzahl der Aufträge, die innerhalb einer bestimmten Zeitspanne ausgeführt werden

Schedulingkriterien

- **Einhaltung von Fristen**: d.h. $\forall i : c_i < d_i$
- Berücksichtigung von Kausalzusammenhängen (Synchronisation, Vorranggraphen Präzedenzsystemen)

Zusätzliche Kriterien können anwendungsabhängig hinzugenommen werden, solange sich nicht den beiden oben genannten Kriterien untergeordnet bleiben.

Scheduling-Verfahren

Pläne

Phasen der Planung:

- Test auf Einplanbarkeit (feasability check)
- Planberechnung (schedule construction)
- Umsetzung auf Zuteilung im Betriebssystem (dispatching)

Gesucht: Plan mit aktueller Start und Endzeit für jeden Prozess P_i.
Darstellung zum Beispiel als nach der Zeit geordnete Liste von Tupeln(P_i, s_i, c_i)
Falls Prozesse unterbrochen werden können, so kann jedem Prozess P_i auch eine Menge von Tupeln zugeordnet werden.

Definitionen:

Zulässiger Plan: Ein Plan ist zulässig, falls alle Prozesse einer Prozessmenge eingeplant sind und dabei keine Präzedenzrestriktionen und keine Zeitanforderungen verletzt werden.

Optimales Planungsverfahren: Ein Verfahren ist optimal, falls es für jede Prozessmenge unter gegebenen Randbedingung einen zulässigen Plan findet, falls ein solcher existiert.

Test auf Einplanbarkeit:

Es können zwei Bedingungen angegeben werden, die für die Existenz eines zulässigen Plans notwendig sind:

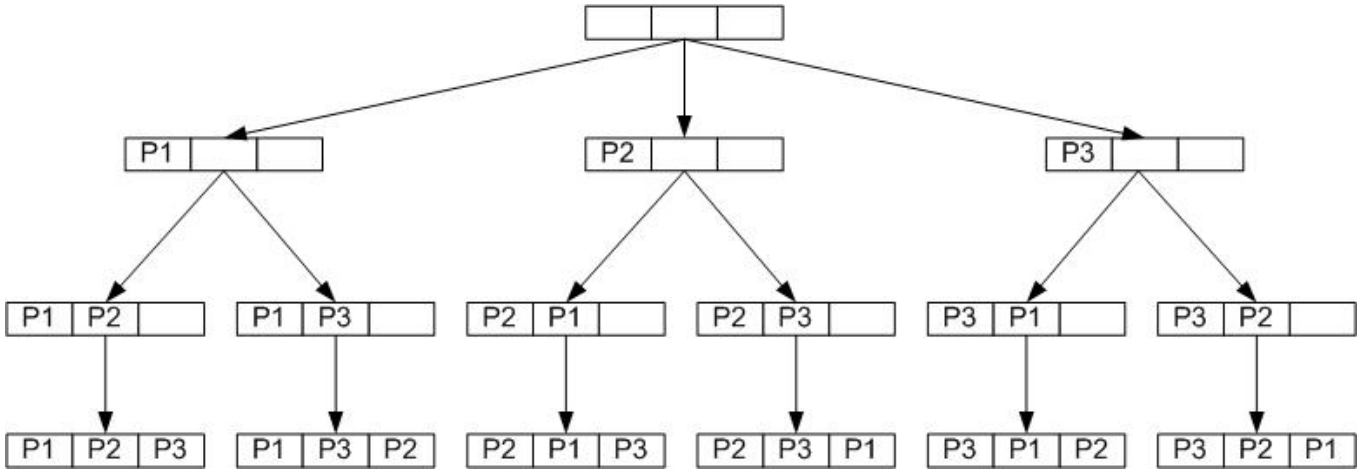
1. $r_i + e_i \leq d_i$, d.h. jeder Prozess muss in dem Intervall zwischen Bereitzeit und Frist ausgeführt werden können.
2. Für jeden Zeitraum $[t_i, t_j]$ muss die Summe der Ausführungszeiten e_x der Prozesse P_x mit $r_x \geq t_i \wedge d_x \leq t_j$ kleiner als der Zeitraum sein.

Schedulingverfahren

- Planen aperiodischer Tasks
  - Planen durch Suchen
  - Planen nach Fristen
  - Planen nach Spielräumen
- Planen periodischer Tasks
  - Planen nach Fristen
  - Planen nach Raten
- Planen abhängiger Prozesse

Planen durch Suchen

- Zunächst ununterbrechbare Aktionen/Prozesse vorausgesetzt
- exakte Planung über Durchsuchen des Lösungsraums
- Beispiel:
  - n=3 Prozesse P_1, P_2, P_3 und 1 Prozessor
  - Suchbaum:



- n! Permutationen müssen bewertet werden, bei Mehrfachbetriebsmitteln (z.B. Mehrprozessorsystem) ist das Problem der Planung NP-vollständig.

Reduzierung der Komplexität

Leichte Verbesserungen der Komplexität können durch:

- Abbrechen von Pfaden bei Verletzung von Fristen
- Verwendung von Heuristiken: z.B. Sortierung nach Bereitstellzeiten r_i

erreicht werden.

Prinzipiell ist aber Planen durch Suchen bei komplexen Systemen nicht möglich.

Planen nach Fristen/Spielräumen

1. Fall: Scheduling auf Einprozessorsystemen

- Einziges zu verplanendes Betriebsmittel ist CPU
- Kriterien zur Prüfbarkeit der Einhaltung aller Zeitbedingungen sind relativ einfach:
  - n Prozesse
  - geordnet nach Fristen $\forall i < n : d_i \leq d_{i+1}$
  - gleiche Bereitzeiten $r_i = 0$

$\Rightarrow$ Es muss gelten: $\forall i : d_i \geq \sum_0^i e_i$

Strategien für Scheduling auf Einprozessorsystemen

1. EDF: Planen nach Fristen (Earliest Deadline First): Der Prozess dessen Frist als nächstes endet erhält den Prozessor.
2. LST: Planen nach Spielraum (Least Slack Time): Der Prozess mit dem kleinsten Spielraum (Deadline-aktuelle Zeit - remaining time) erhält den Prozessor. Der Spielraum eines Prozesses der gegenwärtig ausgeführt wird ist konstant. Die Spielräume aller anderen Prozesse nehmen ab.

Vorteil von LST: LST erkennt Fristverletzungen früher als EDF.

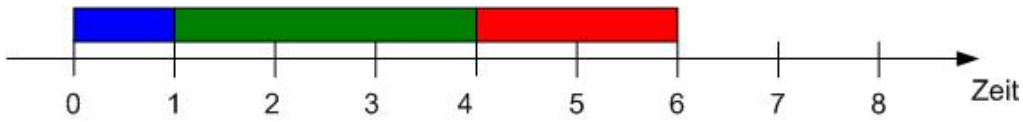
Beispiel

Prozesse:

$P_1 : r_1 = 0; e_1 = 2; d_1 = 8;$

$P_2 : r_2 = 0; e_2 = 3; d_2 = 5;$

$P_3 : r_3 = 0; e_3 = 1; d_3 = 4;$



Earliest Deadline First

Least Slack Time

Optimalität von EDF und LST

Unter der Voraussetzung, dass für alle Prozesse P_i eine Bereitzeit $r_i = 0$ gilt und das ausführende System ein Einprozessorsystem ist, sind die beiden Strategien EDF und LST optimal, d.h. ein zulässiger Plan wird gefunden, falls ein solcher existiert.

Beweisidee für EDF: Tausch in existierendem Plan

- Sei $Plan_X$ ein zulässiger Plan.
- Sei $Plan_{EDF}$ der Plan der durch eine EDF-Strategie erstellt wurde.
- Die Prozessmenge sei nach Fristen geordnet, d.h. $d_i \leq d_j$ für $i < j$.
- Idee: Schrittweise Überführung des Planes $Plan_X$ in $Plan_{EDF}$.
- $P(Plan_X, t)$ sei der Prozess, der von $Plan_X$ zum Zeitpunkt t ausgeführt wird.
- $Plan_X(t)$ ist der bis zum Zeitpunkt t in $Plan_{EDF}$ überführte Plan ($Plan_X(0) = Plan_Z$).

Fortsetzung

- Wir betrachten ein Zeitintervall Δ_t.
- Zum Zeitpunkt t gilt:
  $i = P(Plan_{EDF}, t)$
  $j = P(Plan_X, t)$
- Nur der Fall $j > i$ ist interessant, ansonsten sind beide Pläne bis zum Zeitpunkt t identisch.
  - Fall $j > 1$, es gilt:
  - $d_i \leq d_j$
  - $t + \Delta_t \leq d_i$
  - $Plan_X$ ist ein zulässiger Plan $\Rightarrow$ P_i im $Plan_{EDF}$ rechnet und die Pläne bis zur Zeit t identisch sind, kann P_i auch in $Plan_X$ noch nicht beendet sein, also gilt:
    $\exists t' > t + \Delta_t : i = P(Plan_X, t') = P(Plan_X, t' + \Delta t')$
    $\wedge t' + \Delta t \leq d_i \leq d_j$
  - $\Rightarrow$ Übergang von $Plan_X(t)$ zu $Plan_X(t + t')$ durch Tauschen der Aktivitätsphase von P_i und P_j.
  - Zeitbedingungen werden durch den Übergang nicht verletzt.

Versagen der Strategien bei unterschiedlichen Bereitzeiten

Haben die Prozesse unterschiedliche Bereitzeiten, so versagen die beiden Verfahren:

$P_1 : r_1 = 0; e_1 = 4; d_1 = 8;$

$P_2 : r_2 = 2; e_2 = 2; d_2 = 5;$

EDF- und LST-Verfahren, Deadline d_2 wird verpasst

Zulässiger Plan (nicht präemptiv)

Modifikationen

Durch folgende Modifikationen kann die Optimalität wieder hergestellt werden:

- Präemptive Strategie
- Neuplanung beim Erreichen einer neuen Bereitzeit
- Einplanung nur derjenigen Prozesse, deren Bereitzeit erreicht ist
- Entspricht Neuplanung, wenn ein Prozess aktiv wird
- Bei LST: evtl. Zeitscheiben für Prozesse mit gleichem Spielraum

Zeitplanung bei Mehrprozessorsystemen

Zeitplanung bei Mehrprozessorsystemen

- EDF nicht optimal, egal ob präemptiv oder nichtpräemptive Strategie
- LST geht nur, falls alle Bereitzeitpunkte r_i gleich
- korrekte Zuteilungsalgorithmen erfordern das Abarbeiten von Suchbäumen mit NP-Aufwand oder geeignete Heuristiken
- Beweisidee zur Optimalität von LST bei gleichen Bereitzeitpunkten:Es wird immer am Prozess mit geringstem Spielraum gearbeitet, d.h. wenn dort Zeitüberschreitung auftritt, dann auch, falls noch Prozesse mit größerem Spielraum eingeschoben würden.

Beispiel: Versagen von EDF

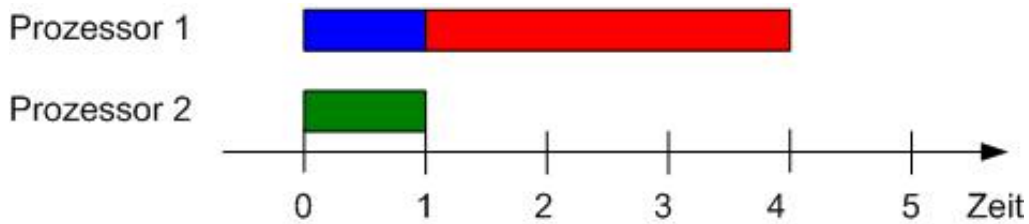
Bei folgendem Beispiel versagt EDF:

2 Prozessoren, 3 Prozesse:

$P_1 : r_1 = 0; e_1 = 3; d_1 = 3;$

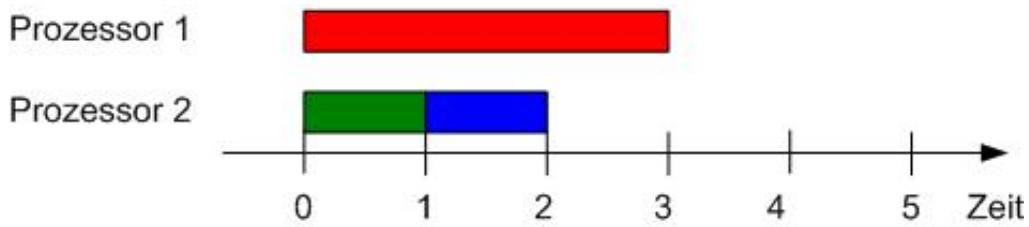
$P_2 : r_2 = 0; e_2 = 1; d_2 = 2;$

$P_3 : r_3 = 0; e_3 = 1; d_3 = 2;$

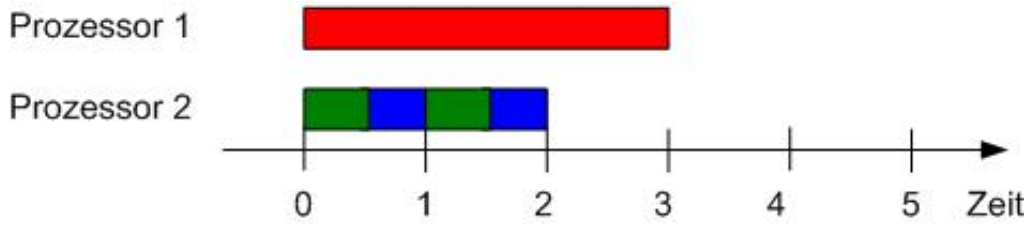


EDF-Verfahren, Deadline d_1 wird verpasst

Optimaler Plan und LST-Verfahren



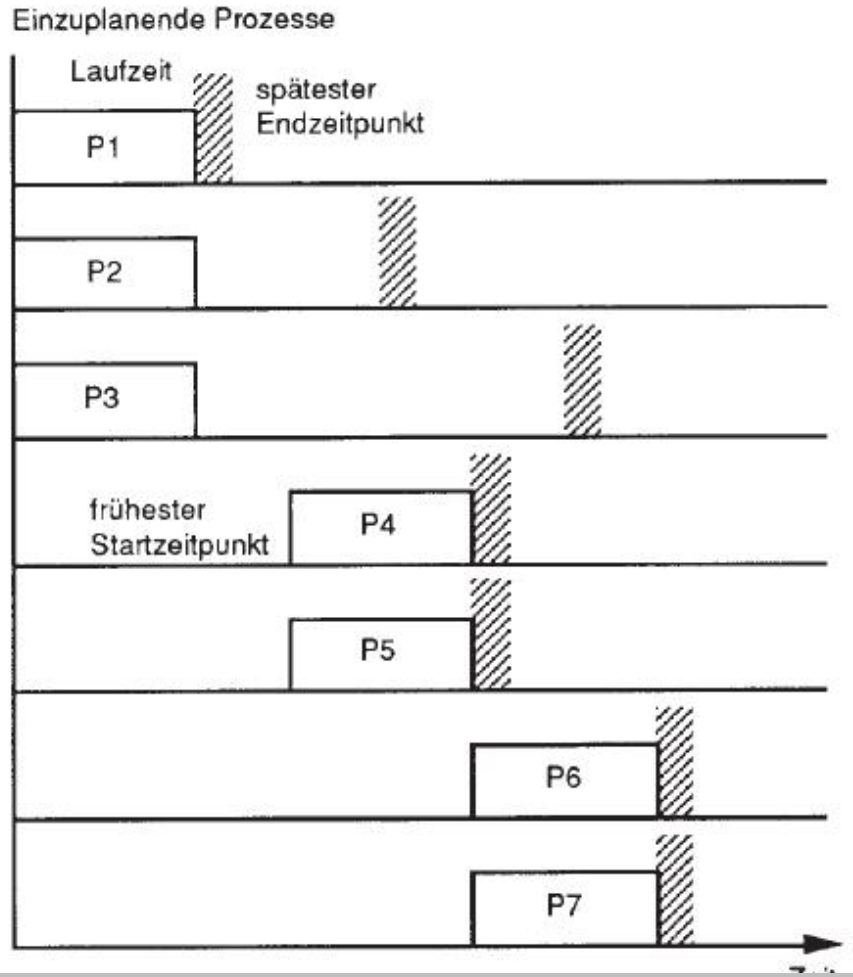
Optimaler Plan



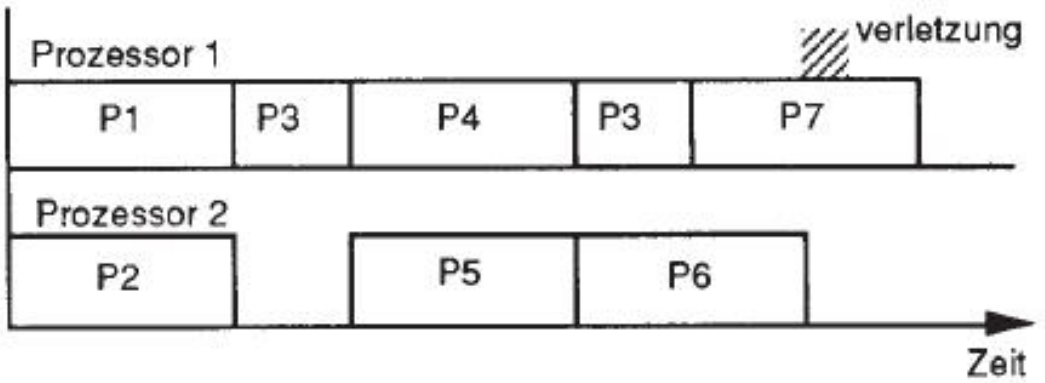
LST-Verfahren mit $\Delta_t = 0.5$

Beispiel: Versagen von LST

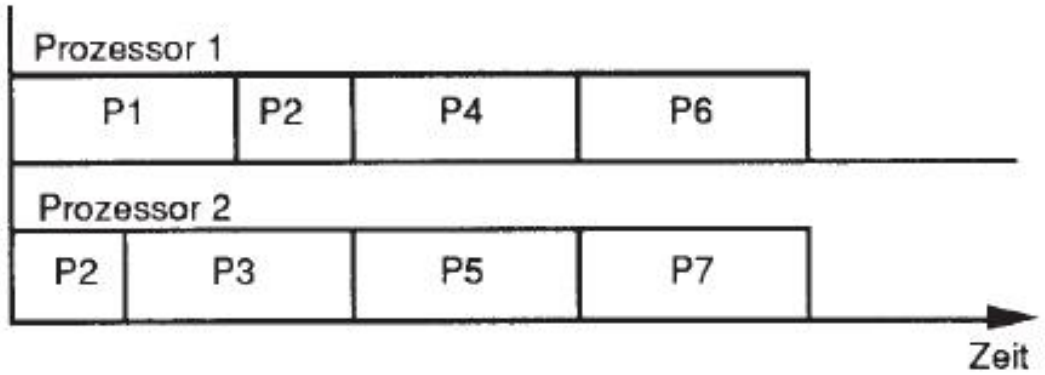
Bei unterschiedlichen Bereitzeiten der Prozesse versagt jedoch auch das LST-Verfahren:



(a) Prozessorvergabe nach Spielraum



(b) Zeitgerechte Prozessorvergabe (nicht algorithmisch ermittelt)



Versagen von Scheduling Verfahren

Alle Verfahren versagen, sobald die Bereitstellzeiten unterschiedlich und nicht von vornherein bekannt sind:
Beweis:

- n CPUs und n-2 Prozesse ohne Spielraum (d.h. die Prozesse müssen sofort rechnen)
- $\Rightarrow$ Problem wird auf 2 Prozessor-Problem reduziert.
- drei weitere Prozesse sind vorhanden und einzuplanen.
- Reihenfolge von Strategie abhängig, alle Reihenfolgen müssen betrachtet werden
- Später treffen weitere Prozesse ein, so dass auf jeden Fall Fristenüberschreitungen auftreten
- Eine Planung wäre aber möglich gewesen, falls alle Bereitzeiten bekannt gewesen wären.

Fortsetzung Beweis

Zahlenwerte:
$P_1 : r_1 = 0; e_1 = 1; d_1 = 1;$
$P_2 : r_2 = 0; e_2 = 2; d_2 = 4;$
$P_3 : r_3 = 0; e_3 = 1; d_3 = 2;$
$\Rightarrow$ Prozess P_1 (kein Spielraum) muss sofort auf CPU1 ausgeführt werden.
$\Rightarrow$ Es gibt je nach Strategie zwei Fälle zu betrachten: P_2 oder P_3 an CPU2

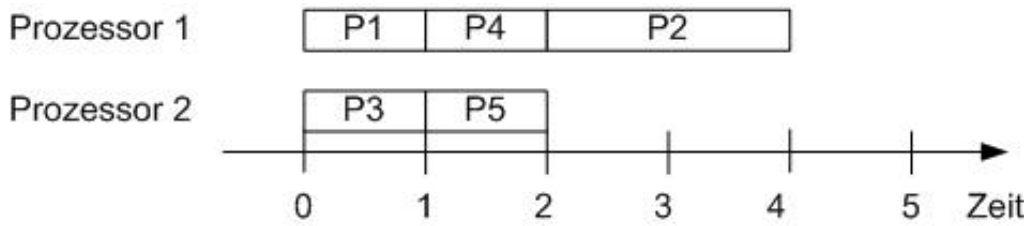
1. Fall

1.Fall: P_2 wird zum Zeitpunkt 0 auf CPU2 ausgeführt.

- Zum Zeitpunkt 1 muss dann P_3 ausgeführt werden.
- Zum Zeitpunkt 1 treffen aber zwei weitere Prozesse P_4 und P_5 mit Deadline 2 und Ausführungsdauer 1 ein.

$\Rightarrow$ Es gibt 3 Prozesse ohne Spielraum.

Aber es gibt einen gültigen Ausführungsplan:



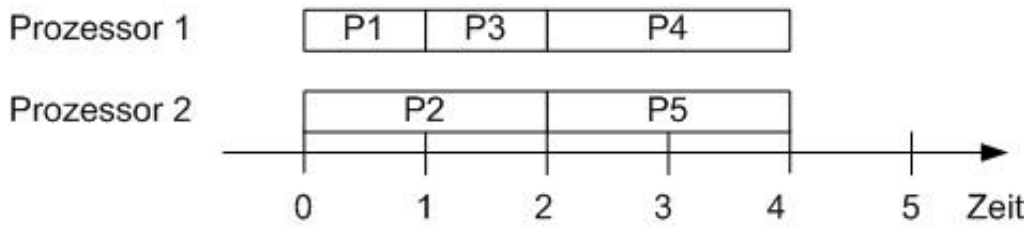
2. Fall

2.Fall: P_3 wird zum Zeitpunkt 0 auf CPU2 ausgeführt.

- Zum Zeitpunkt 1 sind P_1 und P_3 beendet.
- Zum Zeitpunkt 1 beginnt P_2 seine Ausführung.
- Zum Zeitpunkt 2 treffen aber zwei weitere Prozesse P_4 und P_5 mit Deadline 4 und Ausführungsdauer 2 ein.

$\Rightarrow$ Es gibt für 5 nötige Ausführungseinheiten nur 4 vorhandene.

Aber es gibt einen gültigen Ausführungsplan:



Strategien in der Praxis

Die Strategien EDF und LST werden selbst in der Praxis selten angewendet, da:

- kein abgeschlossenes System der Aktionen vorhanden ist (Alarme, Interrupts erfordern dynamischen Planung),
- Bereitzeiten nur bei zyklischen oder Terminprozessen bekannt sind,
- Laufzeiten nicht exakt bekannt sind und
- Synchronisation, Kommunikation und gemeinsame Betriebsmittel die Forderung nach Unabhängigkeit der Aktionen verletzen.

Ansatz in der Praxis

- Zumeist basiert das Scheduling auf der Zuweisung von statischen Prioritäten
- Prioritäten werden meistens durch natürliche Zahlen zwischen 0 und 255 ausgedrückt. Die höchste Priorität kann dabei sowohl 0 (z.B. in VxWorks) als auch 255 (z.B. in POSIX) sein.
- Die Priorität ergibt sich aus der Wichtigkeit des technischen Prozesses und der Abschätzung der Laufzeiten und Spielräume
- bei gleicher Priorität wird zumeist eine FIFO-Strategie angewandt (d.h. ein Prozess läuft solange bis er entweder beendet wird oder aber ein Prozess höherer Priorität eintrifft)

Prozessorvergabe

Grundsätzlich muss beim Scheduling sichergestellt werden, dass Prioritätsinkonsistenzen nur kurz andauern. Der Prozessor kann neu vergeben werden, falls:

- Ein Prozess endet.
- Ein Prozess in den blockierten Zustand wechselt.
- Eine neue Anforderung eintrifft (und ein neuer Prozess gestartet wird)
- Ein Prozess vom blockierten Zustand in den Wartezustand wechselt.
- Nach Ablauf von bestimmten Zeitintervallen (z.B. Round Robin: Prozesse gleicher Priorität werden immer für eine bestimmte Zeitdauer ausgeführt)

$\Rightarrow$ Hochpriorisierte Prozesse dürfen in Echtzeitsystemen nicht durch unwichtigere behindert werden. Daraus folgt eine Prioritätsreihenfolge bei allen Betriebsmitteln (CPU, Semaphore, Netzkommunikation, Puffer, Peripherie), d.h. Vordrängen in allen Warteschlangen.

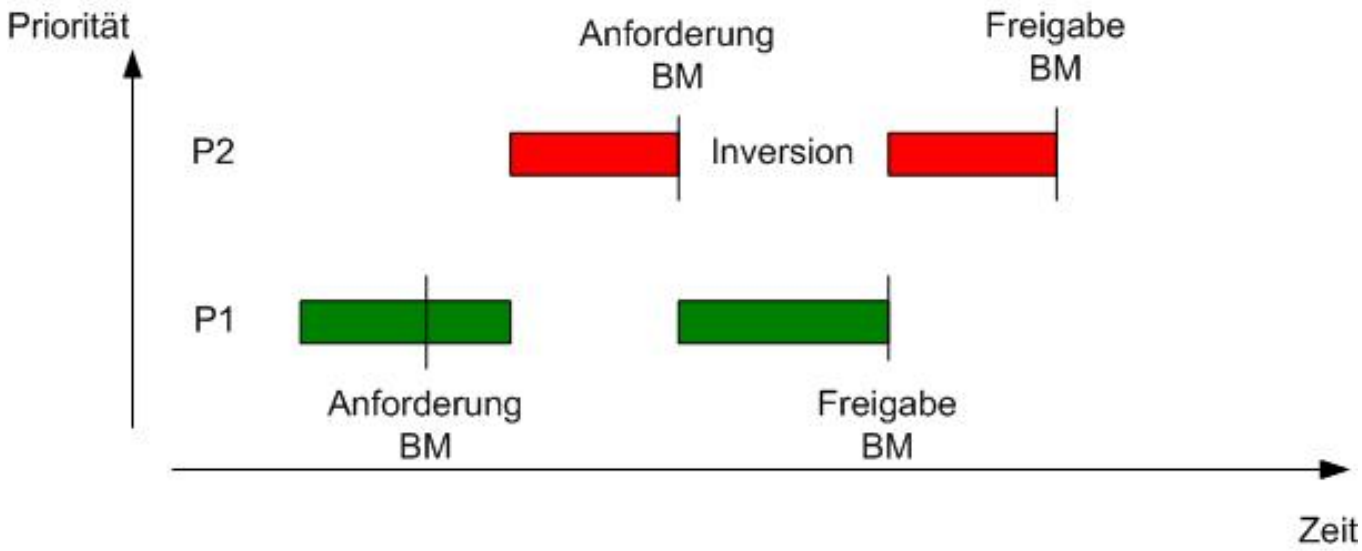
Problem: Prioritätsinversion

Das Problem der **Prioritätsinversion** bezeichnet eine Situation in der ein Prozess mit niedriger Priorität einen Prozess mit hoher Priorität blockiert.

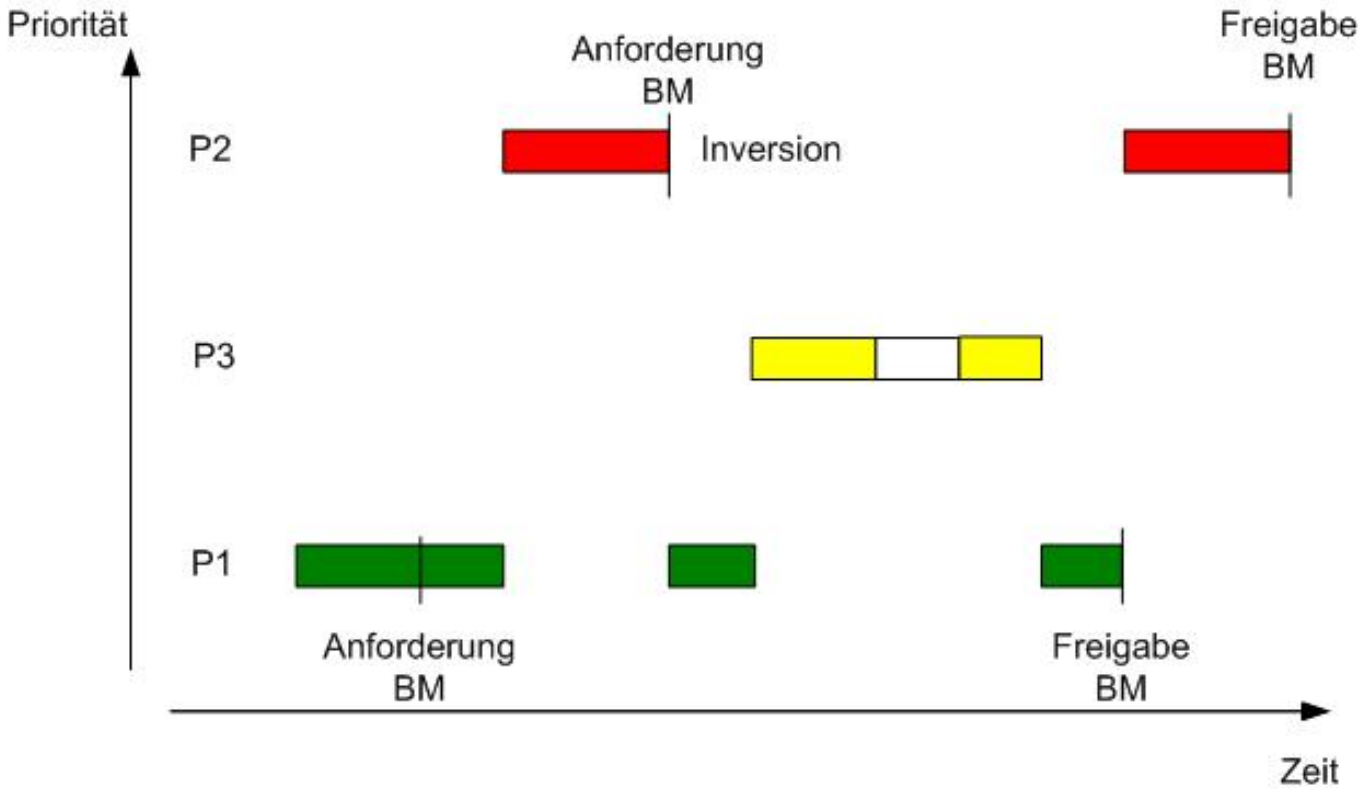
Dabei werden zwei Arten der Prioritätsinversion unterschieden:

- begrenzte (bounded) Prioritätsinversion: die Inversion ist durch die Dauer des kritischen Bereiches beschränkt
- unbegrenzte (unbounded) Prioritätsinversion: der kritische Abschnitt wird durch andere Prozesse auf unbestimmte Zeit blockiert

Begrenzte Inversion



Unbegrenzte Inversion



Reales Problem: Mars Pathfinder

System: Der Mars Pathfinder hatte zur Speicherung der Daten einen Informationsbus (vergleichbar mit Shared Memory). Der Informationsbus war durch einen Mutex (binären Semaphore) geschützt. Ein Bus Management Task verwaltete den Bus mit hoher Priorität.
Ein weiterer Task war für die Sammlung von geologischen Daten eingeplant. Dieser Task lief mit einer niedrigen Priorität. Zusätzlich gab es noch einen Kommunikationstask mittlerer Priorität.

Symptome: Das System führte in unregelmäßigen Abständen einen Neustart durch. Daten gingen dadurch verloren.

Ursache: Der Mutex war nicht mit dem Flag zur Unterstützung von Prioritätsvererbung (siehe später) erzeugt worden. Dadurch kam es zur Prioritätsinversion. Ein Watchdog (Timer) erkannte eine unzulässige Verzögerung des Bus Management Tasks und führte aufgrund eines gravierenden Fehlers einen Neustart durch.

Lösungsansätze für Prioritätsinversion

Zur Lösung der Prioritätsinversion existieren verschiedene Ansätze:

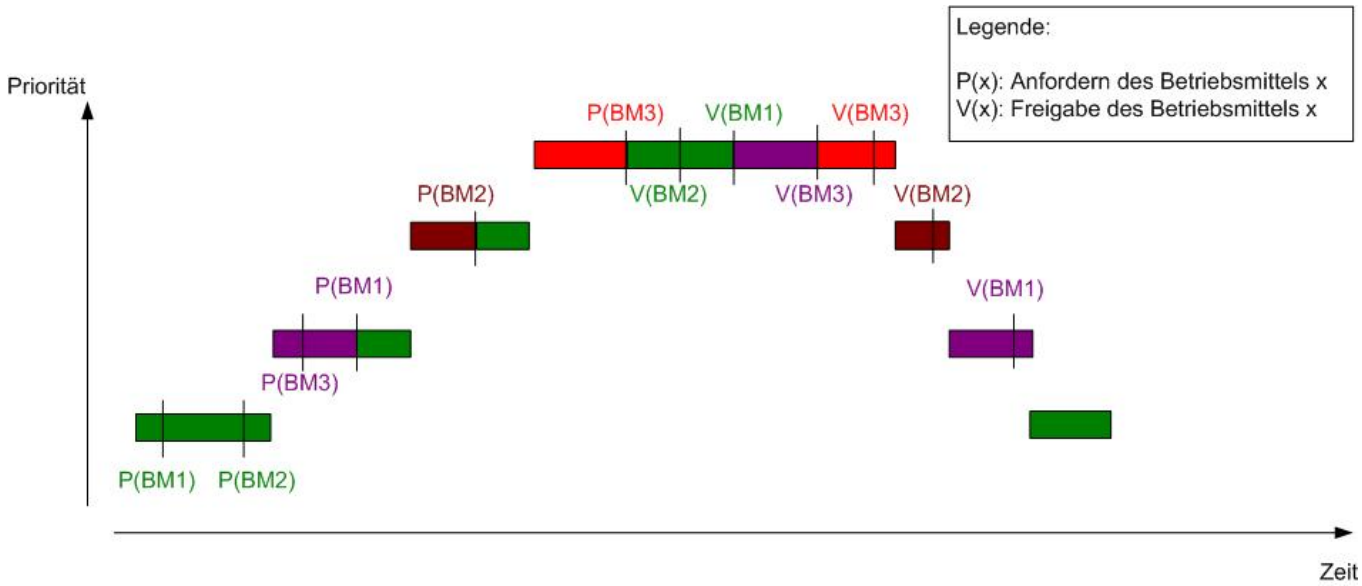
- Prioritätsvererbung (priority inheritance)
- Prioritätsgrenzen (priority ceiling)
- Immediate priority ceiling

Priority inheritance

Der Prozess, der das entsprechende Betriebsmittel besitzt, erbt die höhere Priorität des Prozesses, der das Betriebsmittel anfordert, solange dieser das gemeinsame Betriebsmittel blockiert.

⇒ Verhindert unbegrenzte Blockierung.
⇒ Die Dauer der Blockierung wird auf die Dauer des kritischen Abschnitts begrenzt.
⇒ Die Blockierungen werden hintereinander gereiht (Blockierungsketten).
⇒ Es verhindert keine Deadlocks.

Prioritätsvererbung



Priority ceiling

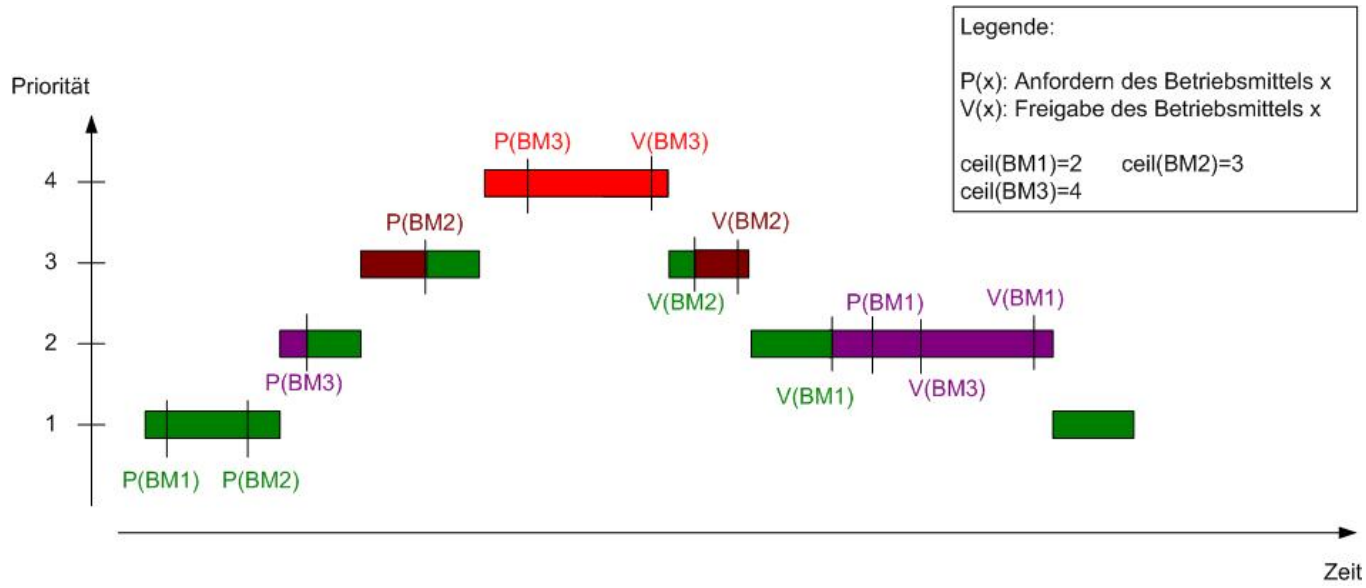
Jedes Betriebsmittel (z.B. Semaphor) s erhält eine Prioritätsgrenze ceil(s), dieses entspricht dem Maximum der Prioritäten der Prozesse, die auf s zugreifen.

- Der Prozess p darf ein BM nur blockieren, wenn er von keinem anderen Prozess, der andere BM besitzt, verzögert werden kann.
- Die aktuelle Prioritätsgrenze für Prozess p ist
  $aktceil(p) = max\{ceil(s)|s \in lockedsem\}$
  $lockedsem$ = Menge aller blockierter BM
- Prozess p darf Betriebsmittel s benutzen, wenn $aktprio(p) > aktceil(p)$
- Andernfalls gibt es genau einen Prozess, der s besitzt. Die Priorität dieses Prozesses wird auf aktprio(p) gesetzt.

$\Rightarrow$ Blockierung nur für die Dauer eines kritischen Abschnitts
$\Rightarrow$ Verhindert Deadlocks
$\Rightarrow$ schwieriger zu realisieren, zusätzlicher Prozesszustand

Vereinfachtes Protokoll: **Immediate priority ceiling**: Prozesse, die ein Betriebsmittel s belegen, bekommen sofort die Priorität $ceil(s)$ zugewiesen.

Prioritätsgrenzen



Zeitplanen periodischer Prozesse

Zeitplanen periodischer Prozesse

Annahmen für präemptives Scheduling:

- Alle Prozesse treten periodisch mit einer Frequenz f_i auf.
- Die Deadline eines Prozesses orientiert sich am Startpunkt des Prozesses.
- Die maximalen Ausführungszeiten e_i sind bekannt.
- Die für einen Prozesswechsel benötigten Zeiten sind vernachlässigbar.
- Alle Prozesse sind unabhängig.

Einplanbarkeit

Eine notwendige Bedingung zur Einplanbarkeit ist die Last:

- Last eines einzelnen Prozesses: $\varrho_i = e_i * f_i$
- Gesamte Auslastung bei n Prozessen: $\varrho = \sum_{i=0}^{n} \varrho_i$
- Bei m Einheiten eines Betriebsmittels ist $\varrho < m$ eine notwendige, aber nicht ausreichende Bedingung.

Zeitplanen nach Fristen

Ausgangspunkt: Wir betrachten Systeme mit einem Betriebsmittel und Fristen der Prozesse, die deren Perioden entsprechen, also $d_i = \frac{1}{f_i}$.

Aussage: Die Einplanung nach Fristen ist optimal.

Beweisidee: Vor dem Verletzen einer Frist ist das BM nie unbeschäftigt.

Zeitplanen nach Raten

Rate Monotonic bezeichnet ein Scheduling-Verfahren mit festen Prioritäten, bei dem sich die Prioritäten $Prio(i)$ proportional zu den Frequenzen f_i verhalten.

$\Rightarrow$ Prozesse mit hohen Raten werden bevorzugt. Das Verfahren ist optimal, falls eine Lösung mit statischen Prioritäten existiert. Verfahren mit dynamischen Prioritäten können allerdings eventuell bessere Ergebnisse liefern.

Liu und Layland haben in einer Worst-Case-Analys gezeigt, dass Ratenplanung sicher erfolgreich ist, falls bei n Prozessen gilt:

$$\varrho \leq varroh_{max} = n * (2^{1/n} - 1)$$
$$lim_{n \to \infty} \varrho_{max} = ln2 \approx 0,69$$

Vergleich Planungsverfahren

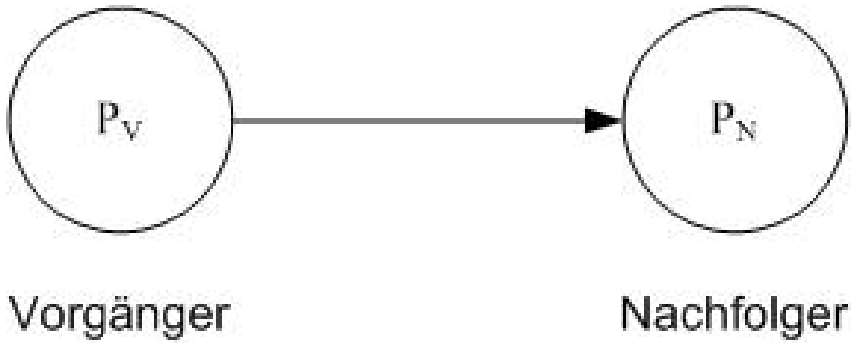
- Einordnung der eingeführten Verfahren
- für Einprozesorsystem (Ausnahme angegeben)
- und statische Planung

Strategie	präemptiv	nichtpräemptiv
Suchen		optimale Pläne O(n!), NP-vollständig
Fristen Spielraum	optimal	optimal bei gleicher Bereitzeit
Spielraum (Mehrpr.)	optimal bei gleicher Bereitzeit	NP-vollständig Anomalien
Monotone Raten	optimal bei beschränkter Auslastung	

Planen abhängiger Prozesse

Präzedenzsysteme

- Präzedenzsysteme beschreiben Folgen von voneinander abhängigen Aktionen.
- Bereitzeiten, Antwortzeiten und Fristen ergeben sich evtl. erst aus dem vorherigen Ablauf von Aktionen
- Typischerweise werden Präzedenzsysteme durch Graphen dargestellt:



Probleme in Präzedenzsystemen

Bei der Planung mit Präzedenzsystemen muss auch berücksichtigt werden, dass Folgeprozess noch rechtzeitig beendet werden können.

Beispiel:
$P_V : r_V = 0,\ e_V = 1,\ d_V{=}3$
$P_N : r_N = 0,\ e_N = 3,\ d_N{=}5$

Falls die Deadline von Prozess P_V gerade noch eingehalten wird, so kann der Prozess P_N nicht mehr rechtzeitig beendet werden.

Lösung: Normalisierung von Präzedenzsystemen

Anstelle des ursprünglichen Präzedenzsystems PS wird ein normalisierten Präzedenzsystem PS' eingeführt. Dieses hat folgende Eigenschaften:

- $\forall i : e'_i = e_i$
- $\forall i : d'_i = \begin{cases} d_i, falls N_i = \emptyset \\ min(d_i, min(d'_q - e'_q | q \in N_i) \end{cases}$
- wobei N_i die Menge der Nachfolger im Präzedenzgraph bezeichnet
- und d'_i rekursiv beginnend bei Prozessen ohne Nachfolger berechnet wird.
- Falls die Bereitzeiten von externen Ereignissen abhängig sind, gilt $r'_i = r_i$. Sind die Bereitzeiten dagegen abhängig von der Beendigung der Prozesse, so ergeben sie die Bereitzeiten aus dem konkreten Scheduling.

$\Rightarrow$ Ein Präzedenzsystem ist nur dann planbar, falls es das zugehörige normalisierte Präzedenzsystem planbar (d.h. es existiert ein gültiger Plan) ist.

Anomalien bei nicht präemptiven Prozessen

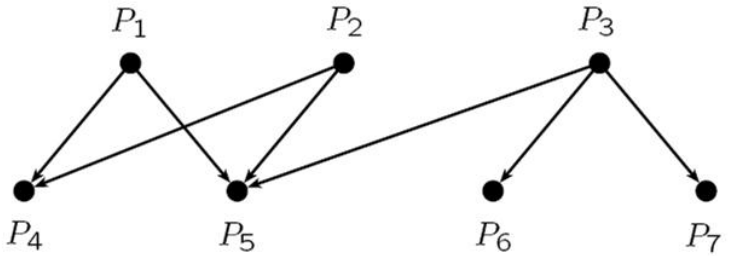
Bei den plausiblen Verfahren EDF und SLT können Anomalien entstehen, falls Prozesse nicht vorzeitig beendet werden:

- Durch Hinzufügen eines Prozessors kann sich die Ausführungszeit verlängern.
- Durch freiwilliges Warten kann die gesamte Ausführungszeit verkürzt werden.

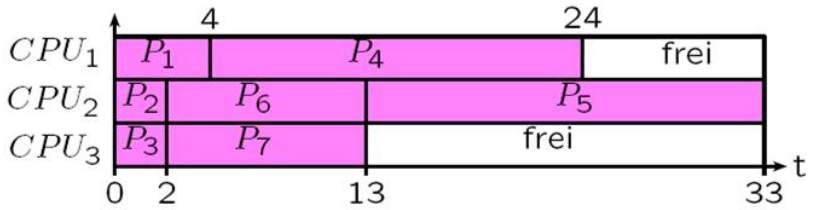
Beispiel: 3 Prozessoren, Präzedenzgraph (siehe nächste Seite), sieben Prozesse:

i	1	2	3	4	5	6	7
e_i	4	2	2	20	20	11	11

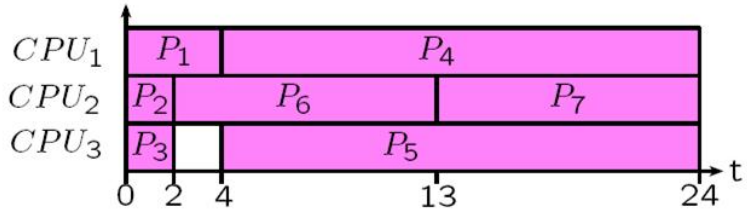
Fortsetzung Anomalie



Präzedenzgraph



Plausible Strategie: Starten falls Vorgänger fertig und Prozessor frei



Optimale Strategie

Exkurs: Worst Case Execution Times

Probleme bei WCET Analyse

Bei der Abschätzung der maximalen Ausführungszeiten stößt man auf einige Probleme:

Es müssen unter anderem die Auswirkungen der Hardwarearchitektur, des Compilers und des Betriebssystems untersucht werden. Dadurch erschwert sich eine Vorhersage.

Zudem dienen viele Eigenschaften der Beschleunigung des allgemeinen Verhaltens, jedoch nicht des Verhaltens im schlechtesten Fall, z.B.:

- Caches, Pipelines, Virtual Memory
- Interruptbehandlung, Präemption
- Compileroptimierungen
- Rekursion

Noch schwieriger wird die Abschätzung falls der betrachtete Prozess von der Umgebung abhängig ist.

Unterscheidungen bei der WCET-Analyse

Bei der Analyse können die Ebenen unterschieden werden:

- Was macht das Programm?
- Was passiert im Prozessor

Zudem werden zwei Methoden unterschieden:

- **statische** WCET Analyse: Untersuchung des Programmcodes
- **dynamische** Analyse: Bestimmung der Ausführungszeit anhand von verschiedenen repräsentativen Durchläufen

Statische Analyse

Aufgaben:

- Bestimmung von Ausführungspfaden in der Hochsprache
- Transformation der Pfade in Maschinencode
- Bestimmung der Laufzeit einzelner Maschinencodesequenzen

Probleme:

- Ausführungspfade lassen sich oft schlecht vollautomatisch ableiten (zu pessimistisch, zu komplex)
- Ausführungspfade häufig abhängig von Eingabedaten

Ansatz: Annotierung der Pfade mit Beschränkungen (wie z.B. maximale Schleifendurchläufe)

Dynamische Analyse

Statische Analysen können zumeist die folgenden Wechselwirkungen nicht berücksichtigen:

- Wechselwirkungen mit anderen Programmen (Stichwort: Wechselseitiger Ausschluss)
- Wechselwirkungen mit dem Betriebssystem (Stichwort: Caches)
- Andere Rechenknoten (Stichwort: Synchronisation)

Durch dynamische Analysen können diese Wechselwirkungen abgeschätzt werden.

Probleme: Wie können sinnvolle Testdurchläufe ausgewählt werden?

Dimensionierung der Rechenleistung

- Aufstellen der Worst-Case Analyse:
  - Bedarf auf bekannten periodischen Anforderungen
  - Bedarf auf erwarteten perdiodischen Anforderunge
  - Zuschlag 100% oder mehr zum Abfangen von Lastspitzen
- Unterschied zu konventionellen Systemen:
  - keine maximale Auslastung des Prozessors
  - keine Durchsatzoptimierung
  - Abläufe sollen determiniert abschätzbar sein

Vorlesung Echtzeitsysteme - Echtzeitbetriebsysteme

Prof. Alois Knoll

Lehrstuhl Embedded Systems and Robotics

Sommersemester 2005

Inhalt

- Grundlagen
- TinyOS (siehe Folien)
- OSEK
- QNX
- VxWorks
- RTLinux/RTAI
- Linux Kernel 2.6
- Windows CE (siehe Folien)

Weitere Echtzeitbetriebssysteme ohne nähere Erläuterung: eCos, LynxOS

Literatur

Literatur:

- OSEK/VDX Operating System, Specification 2.2.3, 2005
- A. Heursch, Time-critical tasks in Linux 2.6, 2005

Links:

- http://www.mnis.fr/en/support/doc/rtos/
- http://aeolean.com/html/RealTimeLinux/RealTimeLinuxReport-2.0.0.pdf
- http://www.osek-vdx.org/
- http://www.qnx.com/
- http://www.windriver.de
- http://www.fsmlabs.com
- http://www.rtai.org
- http://www.tinyos.net/

Anforderungen:

Echtzeitbetriebssysteme unterliegen anderen Anforderungen als Standardbetriebssysteme:

- stabiler Betrieb rund um die Uhr
- definierte Reaktionszeiten
- parallele Prozesse
- Unterstützung von Mehrprozessorsystemen
- schneller Prozesswechsel (geringer Prozesskontext)
- echtzeitfähige Unterbrechensbehandlung
- echtzeitfähiges Scheduling
- echtzeitfähige Prozesskommunikation
- umfangreiche Zeitdienste (absolute, relative Uhren, Weckdienste)
- einfaches Speichermanagment

Fortsetzung

- Unterstützung bei der Ein- und Ausgabe
  - vielfältigste Peripherie
  - direkter Zugriff auf Hardware-Adressen und -Register durch den Benutzer
  - Treiber in Benutzerprozessen möglichst schnell und einfach zu implementieren
  - dynamisches Binden an den Systemkern
  - direkte Nutzung DMA
  - keine mehrfachen Puffer: direkt vom Benutzerpuffer auf das Gerät
- Einfachste Dateistrukturen
- Protokoll für Feldbus oder LAN-Bus, möglichst hardwareunterstützt
- Aufteilung der Betriebssystemfunktionalität in optionale Komponenten (Skalierbarkeit)

Grundlagen

Folgende Aspekte werden wir nun genauer betrachten:

- Schedulingverfahren
- Prozessmanagment
- Speicherbedarf (Footprint)
- Garantierte Reaktionszeiten

Schedulingverfahren

Hier stellen sich folgende Fragen:

- Welche Konzepte sind für das Scheduling von Prozessen verfügbar?
- Gibt es Verfahren für periodische Prozesse?
- Wie wird dem Problem Prioritätsinversion begegnet?
- Wann kann eine Ausführung unterbrochen werden?

Präemptible Betriebssysteme

Betriebssysteme können in drei Klassen unterteilt werden:

- Betriebssysteme mit **kooperativen Scheduling**: es können verschiedene Prozesse parallel ausgeführt werden. Der Dispatcher kann aber einem Prozess den Prozessor nicht entziehen, vielmehr ist das Betriebssystem auf die Kooperation der Prozesse angewiesen (z.B. Windows 95/98/ME)
- Betriebssysteme mit **präemptiven Scheduling**: einem laufenden Prozess kann der Prozessor entzogen werden, falls sich der Prozess im Userspace befindet. (z.B. Linux, Windows 2000/XP)
- **Präemptible** Betriebssysteme: der Prozessor kann dem laufenden Prozess jederzeit entzogen werden, auch wenn sich dieser im Kernel befindet.

$\Rightarrow$ Echtzeitsysteme müssen präemptibel sein.

Prozessmanagement

Bewertung eines Betriebssystems nach:

- Beschränkung der Anzahl von Prozessen
- Möglichkeiten zur Interprozesskommunikation
- Kompatibilität der API mit Standards (z.B. POSIX) zur Erhöhung der Portabilität

Speicherbedarf

Echtzeitbetriebssysteme müssen auf sehr unterschiedlicher Hardware ausgeführt werden. Der verfügbare Speicher variiert sehr stark. Zudem werden viele typische Betriebssystemfunktionalitäten oft gar nicht benötigt (z.B. Dateisysteme, graphische Oberfläche).

$\Rightarrow$ Echtzeitsysteme müssen aus diesen Gründen skalierbar sein, d.h. der Entwickler soll sich die Module auswählen können, deren Funktionalität er für die Anwendung benötigt. Der **minimale Speicherbedarf (Footprint)** ist deshalb oft entscheidend.

Reaktionszeiten

Die Echtzeitfähigkeit wird durch die Messung folgender Zeiten bestimmt:

- Interruptantwortzeiten (interrupt latency): der Zeitraum zwischen dem Auftreten eines Interrupts und der Ausführung des ersten Befehls der dazugehörigen Interruptbehandlungsroutine
- Schedulinglatenz (scheduling latency): Zeit von der Ausführung des letzten Befehls des Interrupt Handlers bis zur Ausführung der ersten Instruktion des Prozesses, der durch das Auftreten des Interrupts in den bereiten Zustand wechselt.
- Zeiten für einen Kontextwechsel (context switch latency): Zeit von der Ausführung des letzten Befehls eines User-Level Prozesses bis zur Ausführung der ersten Instruktion des nächsten User-Level Prozesses.

OSEK

Hintergrund

- Gemeinschaftsprojekt der deutschen Automobilindustrie (u.a. BMW, DaimlerChrysler, VW, Opel, Bosch, Siemens)
- OSEK: Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
- Ziel: Definition einer Standard-API für Echtzeitbetriebssysteme
- Standard ist frei verfügbar (http://www.osek-vdx.org), aber keine freien Implementierungen.
- Es existieren ebenso Ansätze für ein zeitgesteuertes Betriebssystem, sowie eine fehlertolerante Kommunikationsschicht.

Anforderungen

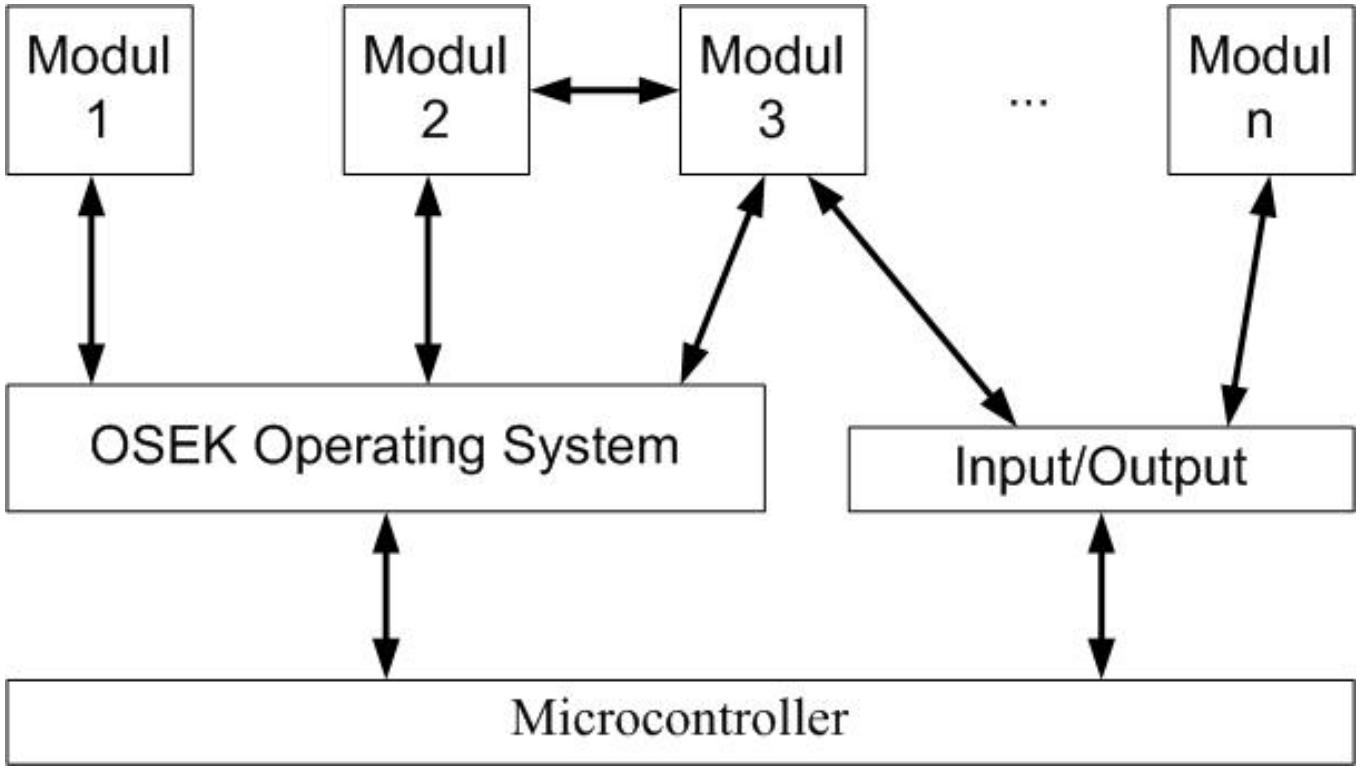
- harte Echtzeitanforderungen
- hohe Sicherheitsanforderungen an Anwendungen
- hohe Anforderungen an die Leistungsfähigkeit
- typischerweise verteilte Systeme
- unterschiedliche Hardware (v.a. Prozessoren)

Ziele

Die Architektur von OSEK wurde durch folgende Ziele beeinflusst:

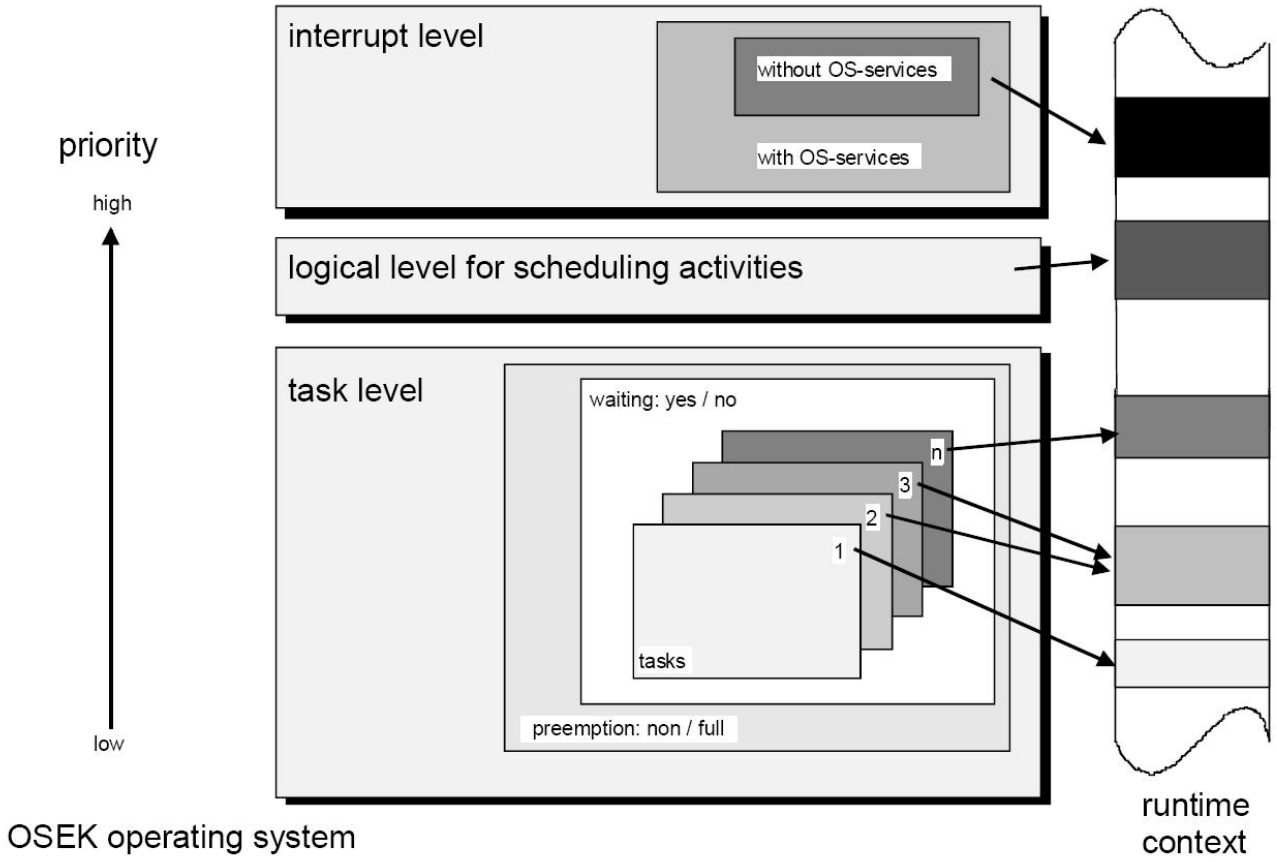
- Skalierbarkeit
- Portabilität der Software
- Konfigurierbarkeit
- Statisches allokiertes Betriebssystem

OSEK Architektur



Die Schnittstelle zwischen Anwendungen und Anwendungsmodulen sind zur Erhöhung der Portierbarkeit standardisiert. Die Ein- und Ausgabe ist ausgelagert und wird auch nicht näher spezifiziert.

Ausführungsebenen in OSEK



Scheduling und Prozesse

Scheduling: Das Scheduling basiert auf einem traditionellen Scheduling mit statischen Prioritäten.

Prozesse: OSEK unterscheidet zwei verschiedene Arten von Prozessen:

1. Basisprozesse
2. Erweiterte Prozesse: haben die Möglichkeit über einen Aufruf der Betriebssystemfunktion waitEvent() auf externe asynchrone Ereignisse zu warten und reagieren.

Der Entwickler kann festlegen, ob ein Prozess unterbrechbar oder ununterbrechbar ist. Es existieren somit vier Prozesszustände in OSEK: running, ready, waiting, suspended.

Betriebssystemklassen

Zum Test auf Übereinstimmung werden Betriebssysteme nach dem OSEK-Standard in vier unterschiedliche Klassen eingeteilt. Die Klassifizierung erfolgt nach der Unterstützung:

1. von mehrmaligen Prozessaktivierungen (einmalig oder mehrfach erlaubt)
2. von Prozesstypen (nur Basisprozesse oder auch erweiterte Prozesse)
3. mehreren Prozessen der selben Priorität

Klassen:

- BCC1: nur einmalig aktivierte Basisprozesse unterschiedlicher Priorität werden unterstützt.
- BCC2: wie BCC1, allerdings Unterstützung von mehrmalig aufgerufenen Basisprozessen, sowie mehreren Basisprozessen gleicher Priorität.
- ECC1: wie BCC1, allerdings auch Unterstützung von erweiterten Prozessen
- ECC2: wie ECC1, allerdings Unterstützung von mehrmalig aufgerufenen Prozessen, sowie mehreren Prozessen gleicher Priorität.

Interruptbehandlung

In OSEK wird zwischen zwei Arten von Interruptbehandlern unterschieden:

- ISR Kategorie 1: Der Behandler benutzt keine Betriebssystemfunktionen. Diese Art von Behandlern sind typischerweise die schnellsten und höchstpriorisierten Unterbrechungen. Im Anschluss der Behandlung wird der unterbrochene Prozess fortgesetzt.
- ISR Kategorie 2: Die Interruptserviceroutine wird durch das Betriebssystem unterstützt, dadurch sind Aufrufe von Betriebssystemfunktionen erlaubt. Falls ein unterbrechenbarer Prozess unterbrochen wurde, wählt der Scheduler nach Beendigung der ISR den nächsten auszuführenden Prozess.

Ressourcenmanagement

Zur Vermeidung von Prioritätsinversion und Deadlocks realisiert OSEK ein Immediate Priority Ceiling Protokoll:

- Jeder Ressource wird eine Grenze (maximale Priorität der die Ressource verwendenden Prozesse) zugewiesen.
- Falls ein Prozess eine Ressource anfordert, wird die aktuelle Priorität des Prozesses auf die entsprechende Grenze angehoben.
- Bei Freigabe kehrt der Prozesse auf die entsprechende Priorität zurück.

QNX

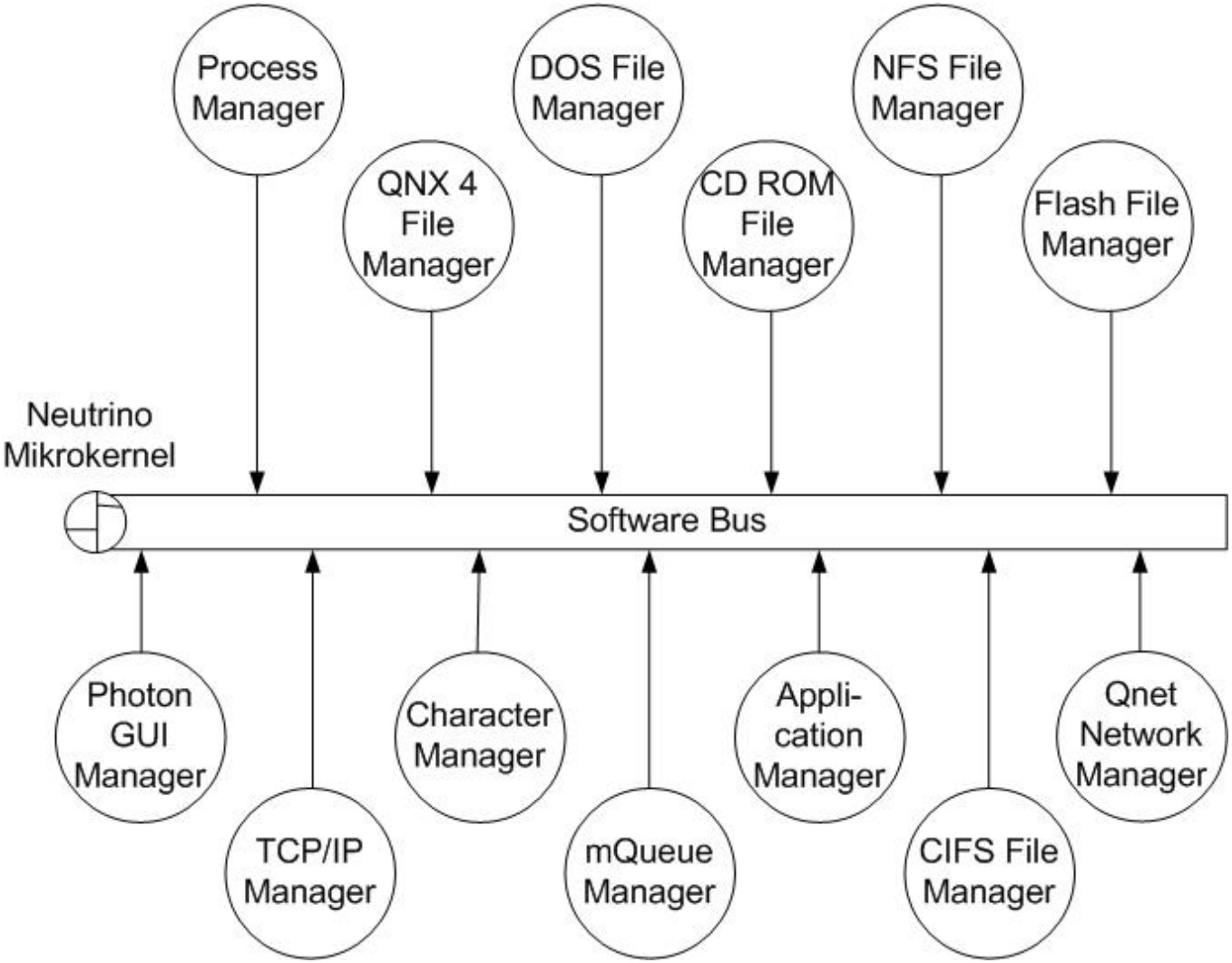
Besonderheiten

- stark skalierbar, extrem kleiner Kernel (bei Version 4.24 ca. 11kB)
- Grundlegendes Konzept: Kommunikation erfolgt durch Nachrichten

Geschichte

- 1980 entwickeln Gordon Bell und Dan Dodge ein eigenes Echtzeitbetriebssystem mit Mikrokernel.
- QNX orientiert sich nicht an Desktopsystemen und breitet sich sehr schnell auf dem Markt der eingebetteten Systeme aus.
- Ende der 90er wird der Kernel noch einmal komplett umgeschrieben, um den POSIX-Standard zu erfüllen. $\Rightarrow$ Ergebnis: QNX Neutrino.

QNX Architektur



Neutrino Microkernel

Der Mikrokernel in QNX enthält nur die notwendigsten Elemente eines Betriebssystems:

- Umsetzung der wichtigsten POSIX Elemente
  - POSIX Threads
  - POSIX Signale
  - POSIX Thread Synchronisation
  - POSIX Scheduling
  - POSIX Timer
- Funktionalität für Nachrichten

Prozessmanager

Als wichtigster Prozess läuft in QNX der Prozessmanager.
Die Aufgaben sind:

- Prozessmanagement: Erzeugen und Löschen von Prozessen, Verwaltung von Prozesseigenschaften
- Speichermanagement: Bereitstellung von Speicherschutzmechanismen, von gemeinsamen Bibliotheken und POSIX Primitiven zu Shared Memory
- Pfadnamenmanagement

Zur Kommunikation zwischen und zur Synchronisation von Prozessen bietet QNX Funktionalitäten zum Nachrichtenaustausch an.

Scheduling

QNX bietet die folgenden Schedulingpolitiken an:

- FIFO Scheduling
- Round Robin Scheduling
- Adaptives Scheduling: Sobald ein Prozess eine Zeitscheibe lang rechnen durfte wird die Priorität um eins gesenkt. Die Priorität sinkt aber nie um mehr als eins unter die Originalpriorität. Sobald ein Prozess blockiert wird, steigt seine Priorität sofort wieder auf die Ursprungspriorität.
- Sporadisches Scheduling: siehe nächste Folie

Die Prioritäten reichen von 0 bis 31 (höchste Priorität). Die Anzahl an möglichen Threads ist dabei unbegrenzt.

Sporadisches Scheduling in QNX

Problem: Gegeben ist ein Prozess fixer Priorität, der auf Ereignisse aus der Umgebung reagieren soll. Tritt nun eine sehr große Anzahl dieser Ereignisse auf, können andere Prozess niedrigerer oder gleicher Priorität evtl. ihre Fristen nicht einhalten.

Lösung: sporadisches Scheduling (vereinfacht)

- Dem Prozess werden zwei Prioritäten zugewiesen: eine Standardpriorität (N) und eine Hintergrundpriorität (L).
- Zusätzlich bekommt der Prozess ein Ausführungsbudget zugewiesen.
- Übersteigt die Ausführungszeit das Budget so wird die Priorität auf L herabgesetzt.
- Nach Ablauf einer vorab festgelegten Periode (replenishment period) wird das Budget wieder erneuert und die Priorität wieder auf N hochgesetzt.

IPC in QNX

Zur Synchronisation bietet QNX folgende Funktionalitäten:

Mechanismus	**Implementiert in**	**Benutzt von**	POSIX
Semaphore	Kernel	Prozesse,Threads	ja
Mutexe	Kernel	Threads	ja
Rekursive Mutexe	Kernel	Threads	ja
Konditionsvariablen	Externer Prozess	Threads	ja
Leser/Schreiber Blockaden	Externer Prozess	Threads	ja
Barriers	Externer Prozess	Threads	ja
FIFO scheduling	Kernel	Prozesse,Threads	nein
Atomare Operationen	Kernel	Prozesse,Threads	nein

IPC in QNX

Zur Kommunikation bietet QNX folgende Funktionalitäten:

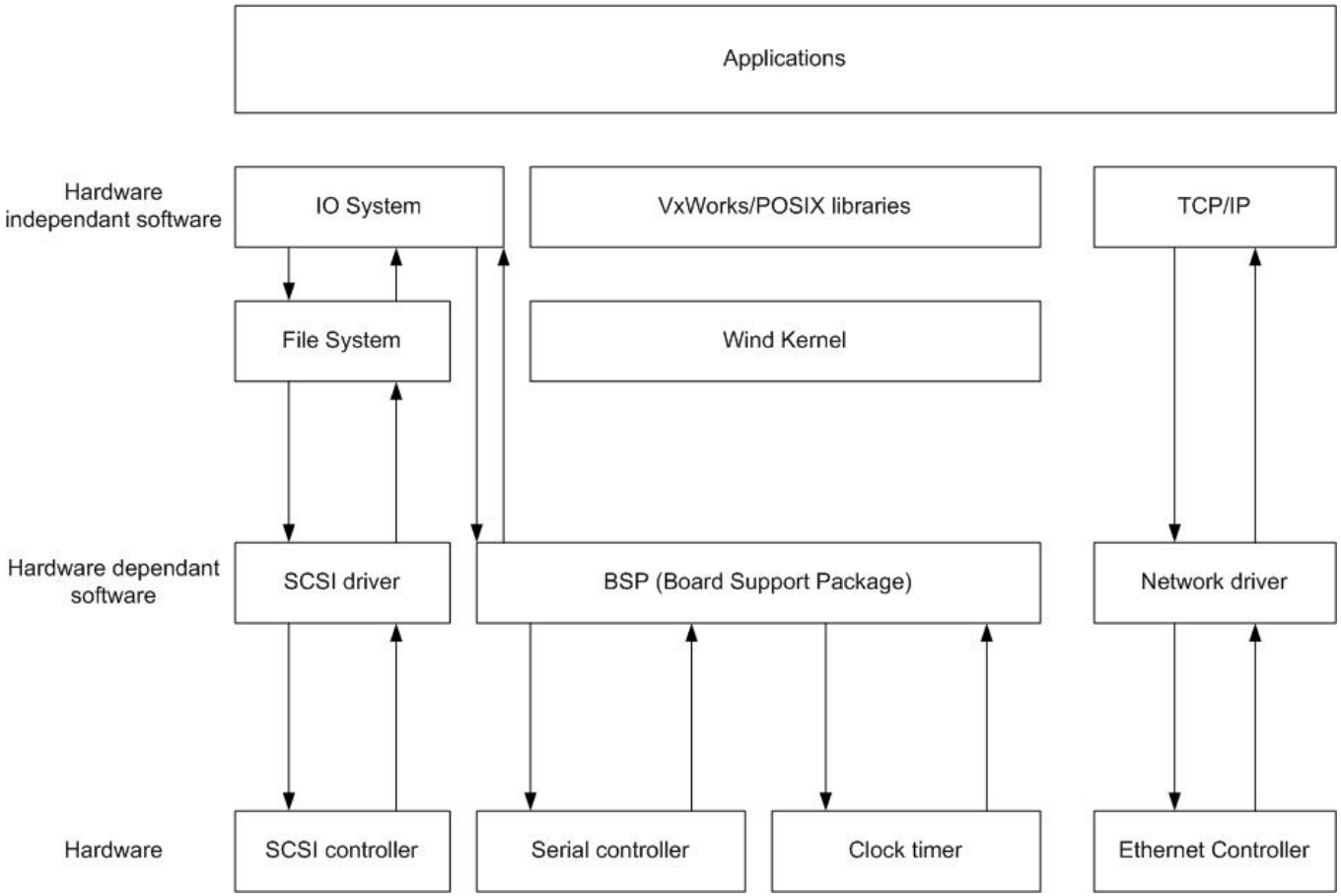
Mechanismus	**Implementiert in**	**Benutzt von**	POSIX
Nachrichten	Kernel	Prozesse, Threads	ja
Signale	Kernel	Prozesse, Threads	ja
Pipes	externer Prozess	Prozesse, Threads	ja
FIFOs	Externer Prozess	Prozesse, Threads	ja
Nachrichtenwarteschlangen	Externer Prozess	Prozesse, Threads	ja
Shared Memory	Prozessmanager	Prozesse, Threads	ja

VxWorks

Eigenschaften

- Host-Target-Entwicklungssystem
- Eigene Entwicklungsumgebung (bis Version 5.5 Tornado, ab 6.0 auf Basis von Eclipse) mit Simulationsumgebung und integriertem Debugger
- Zertifizierbares Echtzeitbetriebssystem

Architektur



Prozessmanagement

- **Schedulingverfahren:** Es werden nur die beiden Verfahren FIFO und RoundRobin angeboten. Ein Verfahren für periodische Prozesse ist nicht verfügbar.
- **Prioritäten:** Die Prioritäten reichen von 0 (höchste Priorität) bis 255.
- **Uhrenauflösung:** Die Uhrenauflösung kann auf eine maximale Rate von ca. 30 KHz (abhängig von Hardware) gesetzt werden.
- **Prozessanzahl:** Die Anzahl der Prozesse ist nicht beschränkt (aber natürlich abhängig vom Speicherplatz)
- **API:** VxWorks bietet zum Management von Prozessen eigene Funktionen, sowie POSIX-Funktionen an.

Speichermanagement

Bis Version 5.5 gab es keinen Schutz des Speichers von Benutzer- und Systemprozessen. Prozesse konnten auf alle Speicherbereiche zugreifen.

Version 6.0 führte Speicherschutz (MMU-Memory Management Unit) ein.

IPC

VxWorks bietet zur Interprozesskommunikation wie bei Prozessen die Funktionalität in einer Windriver eigenen API und einer POSIX-konformen API an.

Unterstützte Konzepte:

- Semaphor
- Mutex (mit Prioritätsvererbung)
- Nachrichtenwartschlangen
- Signale

Neuer Ansatz mit Workbench 2.0

Mit Workbench 2.0 wurde 2005 eine neue Entwicklungsumgebung für Echtzeitanwendungen vorgestellt. Zielplattformen sind Systeme mit den Betriebssystemen VxWorks 6.0 und Linux Kernel 2.4/2.6. Es werden dazu auch modifizierte und zertifizierte Linux Kernel angeboten.
Die Entwicklungsumgebung basiert auf Eclipse.

Linux Kernel 2.6

Bestandsaufnahme

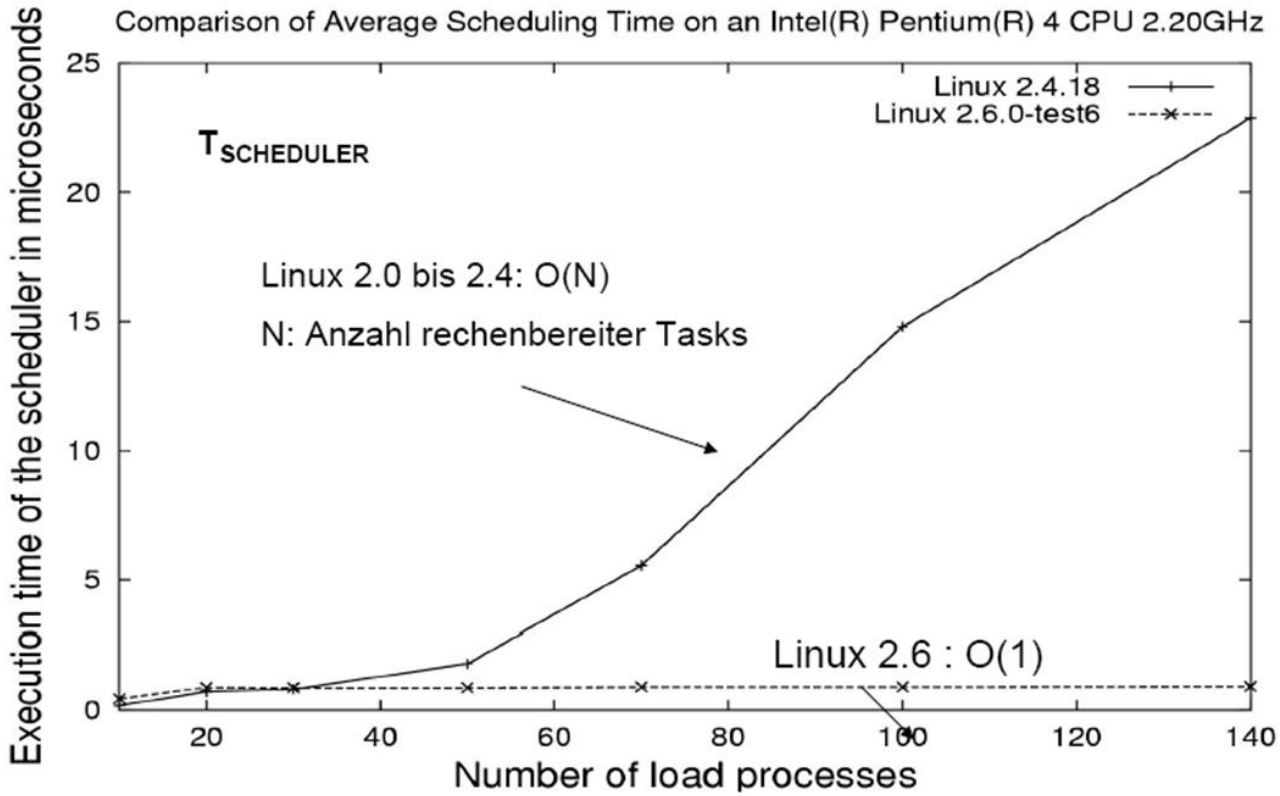
Linux Kernel 2.6 ist geeignet für den Einsatz von Echtzeitbetriebssystemen weil:

- echtzeitfähige Schedulingverfahren vorhanden sind (prioritätenbasiertes Scheduling mit FIFO oder RoundRobin bei Prozessen gleicher Priorität),
- die Zeitauflösung der Uhr auf 1ms (von 10ms in Kernel 2.4) herabgesetzt wurde.

Linux Kernel 2.6 ist nicht geeignet, da:

- der Kernel nicht unterbrechbar ist.

Vergleich Schedulerlaufzeiten 2.4/2.6



(Quelle A. Heursch)

Unterbrechungen des Kernels

Im Kernel ist der **Preemptible Kernel Patch** als Konfigurationsoption enthalten. Dieser erlaubt die Unterbrechung des Kernels.

Allerdings gibt es immer noch eine Anzahl von kritischen Bereichen, die zu langen Verzögerungszeiten führen.

Low Latency Patches helfen bei der Optimierung dieser Latenzzeiten, aber harte Echtzeitanforderungen können nicht erfüllt werden.

Weitere Ansätze siehe Paper von A. Heursch.

Speichermanagement

Linux unterstützt **Virtual Memory**. Dieses Verfahren führt jedoch bei Anwendungen zu zufälligen und nicht vorhersagbaren Verzögerungen, falls sich eine benötigte Seite nicht im Hauptspeicher befindet.

$\Rightarrow$ Echtzeitanwendungen dürfen Virtual Memory nicht verwenden. Linux bietet zur Vermeidung die Funktionen *mlock*() und *mlockall*() an. Dadurch ist es möglich die Auslagerung eines bestimmten Speicherbereiches oder des kompletten Speichers des Prozesses zu verhindern. Dieses Verfahren wird auch mit **Pinning** bezeichnet.

Uhrenauflösung

Die in Linux Kernel 2.6 vorgesehene Uhrenauflösung von 1 ms kann für manche Anwendungen nicht ausreichend sein.

Ist genau festgelegt wann ein Prozess reagieren soll, so kann mittels des **High Resolution Timer Patch** die Auflösung verbessert werden. So kann beispielsweise ein Prozess festlegen, dass ein Interrupt in 3,5 Mikrosekunden von jetzt an erzeugt wird.

Einschränkung: Die zeitliche Angabe muss schon vorab bekannt sein.

RTLinux/RTAI

Ausgangspunkt

Aufgrund von diversen Vorteilen ist eine Verwendung von Linux in Echtzeitsystemen erstrebenswert:

- Linux ist weitverbreitet.
- Treiber sind sehr schnell verfügbar.
- Es stehen viele Entwicklungswerkzeuge zur Verfügung, die Entwickler müssen nicht für ein neues System geschult werden.
- Oftmals müssen nur wenige Teile des Codes echtzeitfähig ausgeführt werden.

Standard-Linux vs. Echtzeit

Leider ist Linux, wie schon gesehen, nicht geeignet für Echtzeitsysteme mit harten Zeitforderungen.

- Linux-Kernel verwendet grobgranulare Synchronisation.
- Trotz gepatchten Kernel oft zu lange Latenzzeiten.
- Höherpriorisierte Tasks können von niederpriorisierten blockiert werden.
- Hintergrund: Hardwareoptimierungsstrategien (z.B. Speichermanagement)

$\Rightarrow$ Linux sollte so modifiziert werden, dass es auch harte Echtzeitforderungen erfüllt.

Möglichkeiten zur Verbesserung von Linux

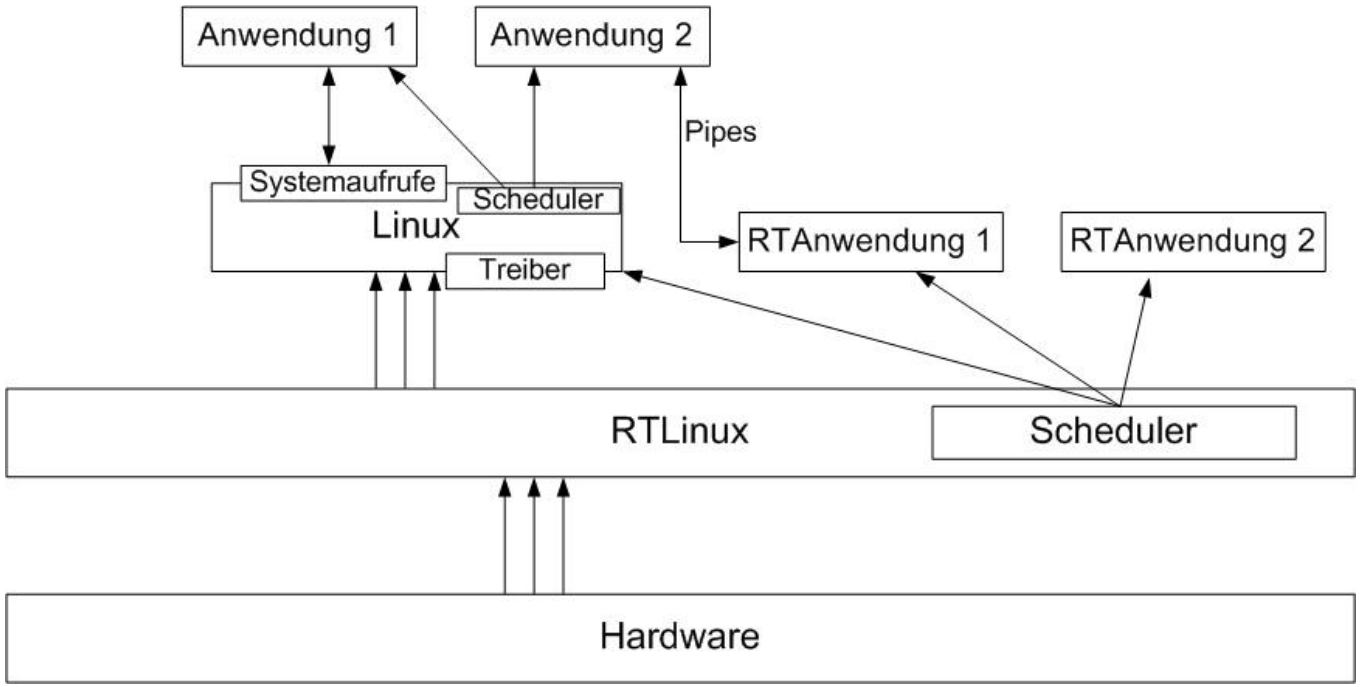
Insgesamt können zwei Ansätze unterschieden werden, Linux zu einem echtzeitfähigen Betriebssystem abzuändern:

1. Verbessern der Unterbrechbarkeit des Kernels (Low Latency Patch, weitere Ansätze siehe Heursch)
2. Einfügen einer neuen Schicht zwischen Hardware und Linux Kernel mit voller Kontrolle über Unterbrechungen $\Rightarrow$ RTLinux, RTAI

Ansatz

- Virtualisierung von Interrupts (Barabanov, Yodaiken, 1996): Interrupts werden in Nachrichten umgewandelt, die zielgerichtet zugestellt werden.
- Virtualisierung der Uhr
- Anbieten von Funktionen zum virtuellen Einschalten (enable) und Ausschalten (disable) von Interrupts.
- Das Linux-System wird als Task niedrigster Priorität ausgeführt.

RTLinux Architektur



Patent

FSMLabs besitzt für RTLinux ein umstrittenes Patent:

A process for running a general purpose computer operating system using a real time operating system, including the steps of:

- providing a real time operating system for running real time tasks and components and non-real time tasks;
- providing a general purpose operating system as one of the non-real time tasks;
- preempting the general purpose operating system as needed for the real time tasks; and
- preventing the general purpose operating system from blocking preemption of the non-real time tasks.

Unterschiede RTAI/RTLinux

- RTLinux verändert Linux-Kernel-Methoden für den Echtzeiteingriff ⇒ Kernel-Versions-Änderungen haben große Auswirkungen.
- RTAI fügt Hardware Abstraction Layer (HAL) zwischen Hardware und Kernel ein. Hierzu sind nur ca. 20 Zeilen Code am Originalkern zu ändern. HAL selbst umfasst kaum mehr als 50 Zeilen ⇒ Transparenz.
- RTAI ist frei, RTLinux in freier (Privat, Ausbildung) und kommerzieller Version.
- Beide Ansätze verwenden ladbare Kernel Module für Echtzeittasks.

Prozessmanagement (RTLinux)

- **Schedulingverfahren:** RTLinux bietet FIFO, EDF und Scheduling von sporadischen Prozessen an.
- **Prioritäten:** Die Prioritäten reichen von 0 bis 1000000 (höchste Priorität).
- **Prozessanzahl:** Die Anzahl der Prozesse ist nicht beschränkt (aber natürlich abhängig vom Speicherplatz), jedoch steigt der Zeitaufwand für das Scheduling proportional mit der Anzahl der Prozesse.
- **API:** Die API von RTLinux ist konform mit dem POSIX-Standard.

Speichermanagment (RTLinux)

Obwohl RTLinux zur Ausführung auf Prozessoren mit MMU-Unterstützung ausgelegt ist, sind die Speicherbereiche der einzelnen Prozesse (auch nicht RTLinux/Linux) nicht getrennt.

Das dynamische Anlegen von Speicher in den echtzeitfähigen Prozessen wird nicht unterstützt. Grund für die nicht vorhandene Unterstützung ist die mangelnde Vorhersagbarkeit bei einer effizienten Implementierung.

IPC (RTLinux)

Zur Prozesssychronisation zwischen echtzeitfähigen Threads werden:

- Semaphoren
- Mutexe (mit immediate priority ceiling)

angeboten.

Zur Kommunikation zwischen Threads (auch zwischen Echtzeit- und Standardthreads) werden:

- shared Memory
- Pipes

angeboten.

Programmierung

RTAI-(RTLinux-)Echtzeitmodule müssen typische Struktur eines Kernel Modulsbesitzen, d.h. insb. die Funktionen:

```
int init_module(void){
 /* Aufgerufen bei insmod*/
 ...
 }

 void cleanup_module(void)
 {
  /* Aufgerufen bei rmmod*/
  ...
 }
```

zur Initalisierung bzw. dem Aufräumen.

Zusammenfassung

- Die meisten Echtzeitbetriebssysteme orientieren sich am POSIX Standard.
- Der minimale Speicherbedarf reicht von wenigen Kilobyte (TinyOS, QNX) bis hin zu mehreren Megabyte (Windows CE / XP Embedded).
- Viele der Betriebssysteme sind skalierbar. Zur Änderung des Leistungsumfangs von Betriebssystemen muss das System entweder neu kompiliert werden (VxWorks) oder neue Prozesse müssen nachgeladen werden (QNX).
- Die Echtzeitfähigkeit von Standardbetriebssysteme kann durch Erweiterungen erreicht werden (RTLinux/RTAI).
- Die Schedulingverfahren und die IPC-Mechanismen orientieren sich stark an den in POSIX vorgeschlagenen Standards.
- Das Problem der Prioritätsinversion wird zumeist durch Prioritätsvererbung gelöst.

Vorlesung Echtzeitsysteme - Programmiersprachen

Prof. Alois Knoll

Lehrstuhl Embedded Systems and Robotics

Sommersemester 2005

Organisatorisches

- Prüfungsanmeldung:
  - Die Anmeldung erfolgt per Email an buckl@in.tum.de
  - Die Prüfungstermine werden voraussichtlich in 1-2 Wochen zugeteilt. Terminwünsche werden soweit möglich berücksichtigt.
- Exkursion
  - Datum. 7.7.05
  - Treffpunkt 9:55 Uhr vor Ort (siehe Beschreibung auf Homepage)

Inhalt

- Motivation
- PEARL
- Ada
- C mit Erweiterungen (z.B. POSIX)
- Real-Time Java

Literatur

- Ascher Opel, Requirements for Real-Time Languages, 1966
- N. Wirth, Embedded Systems and Real-Time Programming, 2001
- Alan Burns, Andy Wellings: Real-Time Systems and Programming Languages, 2001

Links:

- http://www.irt.uni-hannover.de/pearl/
- http://www.ada-deutschland.de/
- http://www.rtj.org/

Anforderungen an Programmiersprachen

Anforderungen an Programmiersprachen

Die Anforderungen an Programmiersprachen für Echtzeitsysteme fallen in verschiedene Bereiche:

- Beherrschung komplexer und nebenläufiger Systeme
- Einhaltung zeitliche Anforderungen
- Möglichkeiten zur hardwarenahen Programmierung
- Erfüllung hoher Sicherheitsanforderungen
- Möglichkeiten zum Umgang mit verteilten Systemen

Beherschung komplexer nebenläufiger Systeme

- Aufteilung der Anwendung in kleinere, weniger komplexe Subsysteme
- Unterstützung von Nebenläufigkeit (Prozesse, Threads)
- Daten- und Methodenkapselung in Modulen
- Eignung für unabhängiges Implementieren, Übersetzen und Testen von Modulen durch verschiedene Personen

Einhaltung zeitlicher Anforderungen

- projektierbares Zeitverhalten
- Prioritäten
- wenig (kein) Overhead durch Laufzeitsystem
- umfangreiche Zeitdienste
- Zeitüberwachung aller Wartezustände
- Aktivierung von Prozessen
  - sofort
  - zu bestimmten Zeitpunkten
  - in bestimmten Zeitabständen
  - bei bestimmten Ereignissen

Möglichkeiten zur hardwarenahen Programmierung

- Ansprechen von Speicheradressen z.B. z.B. „memory mapped“E/A
- Interrupt- und Ausnahme-Behandlung
- Unterstützung vielseitiger Peripherie
- Definition virtueller Geräteklassen mit einheitlichen Schnittstellen
- einheitliches Konzept für Standard- und Prozess-E/A

Erfüllung hoher Sicherheitsanforderungen

- Lesbarkeit, Übersichtlichkeit, Einfachheit durch wenige Konzepte
- Modularisierung und strenge Typüberprüfung als Vorraussetzung zur frühen Fehlererkennung durch Übersetzer, Binder und Laufzeitsystem
- Überprüfbare Schnittstellen (-beschreibungen) der Module
- Verifizierbarkeit von Systemen

Sicherheit fängt schon im Kleinen an

Selbst lexikalische Konventionen können Fehler verhindern.
Negatives Beispiel FORTRAN: Leerzeichen werden bei Namen ignoriert.

Schon ein unterschiedliches Zeichen kann einen völlig neuen Sinn ergeben:

```
DO 20 I=1,100 (Schleife von 1 bis 100)

DO20I=1.100 (Zuweisung von 1.1 an die Variable DO20I)
```

Anmerkung: Variablen müssen in Fortran nicht explizit definiert werden.

Unterstützung der Kommunikation

- vielseitige Protokolle zur Kommunikation (Feldbus-Protokolle, LAN-Protokolle)
- schnelle Kommunikation
- Synchronisation (auch in verteilten Systemen)

Funktionen in verteilten Systemen

- Operationen auf Daten anderer Rechner im Netz
- Objektverwaltung im Netz
- Konfigurierungsfunktionen zur Zuordnung von Programmen/Modulen zu Rechnern
- automatische Rekonfigurierung in Fehlersituationen

Konsequenzen: Benötigte Datentypen

Aus den aufgezählten Anforderungen ergibt sich der Bedarf an speziellen Datentypen:

- Prozesse (task, thread)
- Ausnahmen
- Semaphore, Ereignisse
- Bits, Bytes (z.B. für Statusregister)
- Uhren, Zeit, Zeitdauer

Offene Fragen

- Bei Ausnahmen: Wo wird nach der Behandlung fortgesetzt?
- Soll es untergeordnete Prozesse (Hierarchien) geben?
- Darf ein Prozess neu gestartet werden, falls die vorhergehende Inkarnation noch aktiv ist?
- Darf ein Prozess einen anderen löschen und wenn ja in welchem Zustand?

Geschichte

Geschichte

- 1960-1970
  - Assemblerprogramme da Speicher teuer
  - Programme sind optimiert $\Rightarrow$ jedes Bit wird genutzt
- ab ca. 1966
  - erster Einsatz von höheren Sprachen, z.B.
    - CORAL und RTL/2
    - ALGOL 60
    - FORTRAN IV
  - Prozeduraufrufe für Betriebssystem-Echtzeit-Dienste
  - Probleme:
    - viel Wissen über Betriebssystem nötig
    - wenig portabel
    - keine semantische Prüfung der Parameter durch den Übersetzer, da keine speziellen Datentypen für task, semaphor, clock
    - schwierige Fehlersuche

Geschichte: ab 1970 erste Echtzeitsprachen

- nationale und internationale Normen
  - PEARL (Deutschland): Process and Experiment Automation Realtime Language
  - HAL/S (USA)
  - PROCOL (Japan)
  - RT-FORTRAN
  - RT-BASIC
- neue Datentypen (z.B. task, clock, duration, sema, interrupt) mit zugehörigen Operationen in die Sprache integriert
- einheitliche Anweisungen für EA und die Beschreibung von Datenwegen
- Vorteile:
  - Benutzerfreundliche Sprachelemente
  - Semantik der Parameter bei Betriebssystemaufrufen durch Übersetzer prüfbar
  - weitgehend portabel
- Nachteile: 2 Alternativen
  1. Eigenes umfangreiches Betriebssystem nötig $\Rightarrow$ hohe Entwicklungskosten
  2. Anpassung eines vorhandenen universellen Betriebssystem $\Rightarrow$ Gefahren: ineffizient, überflüssige Teile im Betriebssystem, eingeschränkte Portabilität

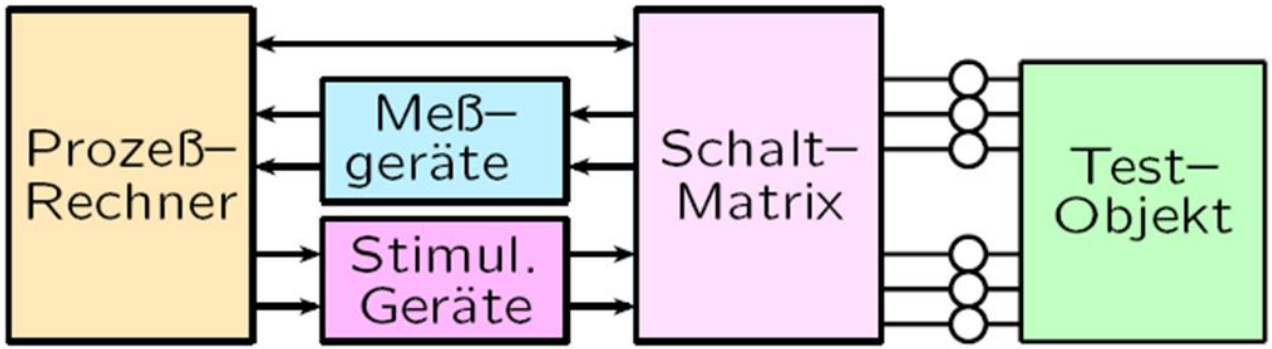
Geschichte ab 1978

- universelle sichere hohe Sprachkonzepte für alle Anwendungsbereiche
- Standardisierung, inbesondere durch Department of Defense (DOD): Ada
- Datentypen (z.B. task, duration, interrupt) oder systemabhängige Parameter werden in sprachlich sauberer Weise mittels Module /Packages eingebunden
- Beispiele:
  - Ada83,Ada95
  - CHILL
  - MODULA.2
  - PEARL, PEARL 90, Mehrrechner.PEARL

Geschichte heute

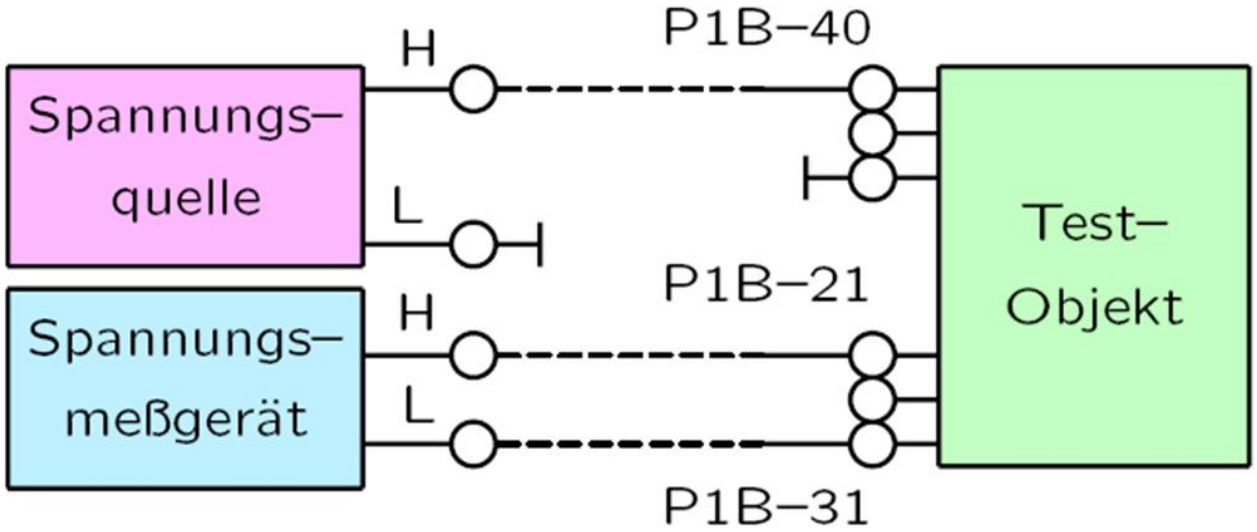
- Trend hin zu universellen Sprachen (z.B. C,C++ oder Java) mit Bibliotheksprozeduren für Echtzeitdienste angereichert
- herstellerspezifische Speziallösungen für eingeschränkte Anwendungsbereiche, z.B.
  - Prüfsysteme
  - Standardregelungsaufgaben
  - Förderungstechnik
  - Visualisierung (Leitstand)
  - Telefonanlagen
- Beispiele:
  - SPS-Programmierung (Speicherprogrammierbare Steuerung)
  - ATLAS (Abbreviated Test Language for All Systems) für Prüfsysteme (v.a. Flugzeugelektronik)
  - ESTEREL
  - POSIX

Beispiel: ATLAS



Blockschaltbild Testsystem

Beispiel: ATLAS



Prüfung Elektronikgerät

Beispiel: ATLAS

- Verbale Testvorschrift:
  - Lege 28V an P1B-40
  - OK, falls 30mV zw. P1B-21 und P1B-31

- ATLAS Codeausschnitt

```
100 302 APPLY, SOURCE, DC SIGNAL, VOLTAGE 28
        ERRLMT + - 0.5V, CNX HI P1B-40 LO COMMON $
100 304 VERIFY, (VOLTAGE), DC SIGNAL, NOM 0.000V
        LL-0.030V UL 0.030V, CNX HI P1B-21 LO P1B-31 $
100 306 GOTO, STEP 105 084, IF NOGO $
...
105 084 PRINT, MESSAGE, 'FEHLER' $
```

PEARL

Hintergrund

- Process and Experiment Automation Real-Time Language
- DIN 66253
- portable Hochsprache für Echtzeitsysteme
- lauffähig z.B. unter UNIX, OS/2, Linux
- Versionen: Basic PEARL (1981), Full PEARL (1982), Mehrrechner PEARL (1988), PEARL 90 (1998)

Eigenschaften

- strenge Typisierung
- modulbasiert
- unterstützt (prioritätenbasiertes) Multitasking
- E/A-Operationen werden von eigentlicher Programmausführung separiert
- Synchronisationsdienste: Semaphore, Bolt-Variablen
- Zugriff auf Interrupts
- erleichterte Zeitverwaltung

Grundstruktur

```
/*Hello World*/
MODULE Hello;

   SYSTEM;
      termout: STDOUT;

   PROBLEM;
      DECLARE x FLOAT;
      T: TASK MAIN;
      x := 3.14; !PI
      PUT 'Hello' TO termout;
   END;

MODEND;
```

Erläuterung

- Anwendungen werden in Module aufgeteilt
- jedes Modul enthält einen Systemteil und einen Problemteil
- Systemteil: Definition von virtuellen Geräten für alle physischen Geräte, die das Modul benutzt
- Probemteil: eigentlicher Programmcode, Deklarationen
- sonstige Notationen typisch: Kommentare, Semikolon zur Terminierung von Anweisungen

Datentypen

Schlüsselwort	**Bedeutung**	**Beispiel**
FIXED	ganzzahlige Variable	-2
FLOAT	Gleitkommazahl	0.23E-3
CLOCK	Zeitpunkt	10:44:23.142
DURATION	Zeitdauer	2 HRS 31 MIN 2.346 SEC
CHAR	Folge von Bytes	'Hallo'
BIT	Folge von Bits	'1101'B1

Datentypen

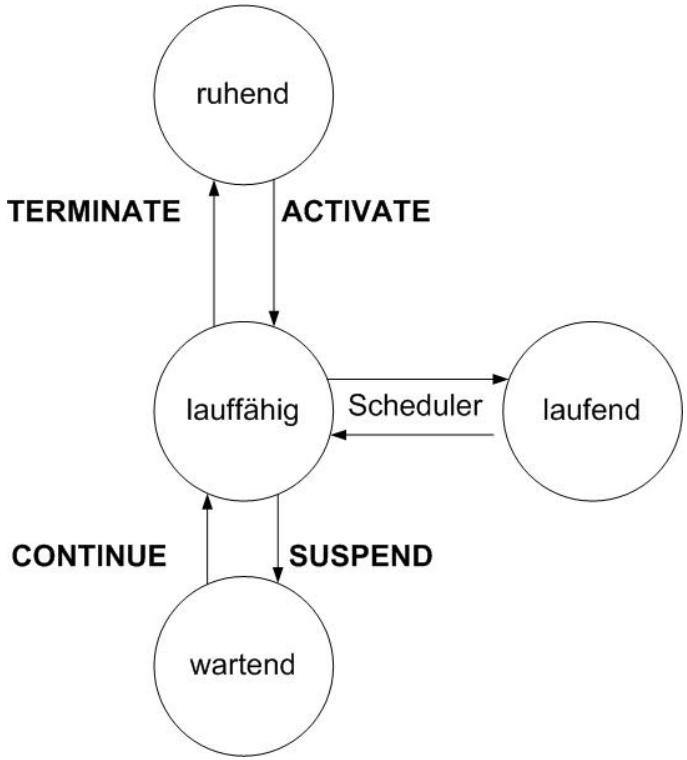
- Variablen werden durch DECLARE deklariert
- Mittels INIT können Variablen initialisiert werden
- Durch das Schlüsselwort INV können Konstanten gekennzeichnet werden
- Die temporalen Variablen bieten eine Genauigkeit von einer Millisekunde
- Die Genauigkeit der Datentypen kann angegeben werden
- Zeiger auf Datentypen werden unterstützt

Prozessmodell

- initial sind alle Prozesse ruhend bis auf MAIN
- Zustandswechsel sind terminiert möglich:

```
AFTER 5 SEC ACTIVATE Task1;
```

```
AT 10:15:0 ALL 2 MIN
   UNTIL 11:45:0
   ACTIVATE Student;
```



Scheduling

- Scheduling ist präemptiv
- Prozessen werden feste Prioritäten zugewiesen (im Task-Kopf)
- Bei Prozessen gleicher Priorität wird Round Robin angewandt
- Die Zeitscheibenlänge ist dabei jeweils abhängig vom Betriebssystem

Synchronisation

Zur Synchronisation bietet PEARL Semaphore und Bolt-Variablen:

Semaphore:

- Deklaration wie bei einer Variablen, Datentyp: SEMA
- Operationen REQUEST und RELEASE zum Anfordern und Freigeben des Semaphores
- Mittels der Operation TRY kann versucht werden den Semaphore nicht blockierend zu lesen
- Es werden keine Möglichkeiten zur Vermeidung von Prioritätsinversion geboten

Bolt-Variablen

- Bolt-Variablen besitzen wie Semaphoren die Zustände belegt und frei und zusätzlich einen 3. Zustand: Belegung nicht möglich
- Schlüsselwort zur Deklaration BOLT
- RESERVE und FREE funktionieren analog zu Semaphore-Operationen REQUEST bzw. RELEASE
- exklusive Zugriffe mit RESERVE haben Vorrang von (nicht exklusiven) Zugriffen mit ENTER
- Eine elegante Formulierung des Leser-Schreiber-Problems ist damit möglich

Interrupt-Behandlung

- Es können nur Prozesse aktiviert oder fortgesetzt werden, die durch WHEN eingeplant sind
- Es ist möglich Interrupts durch DISABLE/ENABLE zu Sperren und Freizugeben

Beispiel: Interrupts

```
MODULE Vorlesung:

   System;
   Weckruf: IR*2;

   PROBLEM;
   SPC Weckruf INTERRUPT;

   Student: TASK PRIORITY 20
   When Weckruf Activate Student1;
   DISABLE Weckruf;
   ...
   ENABLE Weckruf;
   END;

MODEND;
```

Ada

Geschichte

- 1970 von Jean Ichbiah (Firma Honeywell Bull) entworfen
- durch das Department of Defense gefördert
- ist Mitglied der Pascal Familie
- wird häufig für Systeme mit hohen Anforderungen an die Sicherheit verwendet
- Versionen: Ada 83, Ada 95
- freie Compiler sind verfügbar: z.B. http://www.adahome.com

Eigenschaften

- strenges Typsystem
- zahlreiche Prüfungen zur Laufzeit: z.B. zur Erkennung von Speicherüberlaufen, Zugriff auf fremden Speicher, Off-by-One Fehlern
- Ausnahmebehandlung
- generische Systeme

Ab Ada 95:

- objektorientierte Programmierung
- dynamische Polymorphie

Grundstruktur

Strukturierung

Die Programme werden aufgeteilt in

- Blöcke
- Unterprogramme
- Pakete
- Tasks

Die Schachtelung kann dabei beliebig sein. Das Hauptprogramm ist eine Prozedur (main task).

Strukturierung

- Pakete und Tasks müssen, Unterprogramme können in eine Spezifikation (head) und einen Rumpf (body) aufgeteilt werden
- im Kopf stehen die sichtbaren (Export-) Objekte
- die Angabe private spezifiziert lokale Objekte, deren Realisierung verborgen bleiben (nicht ausnutzbar)
- der Anweisungsteil des Pakets wird einmalig beim Ausarbeiten der Paketdeklaration ausgeführt

Syntax Paket

```
PACKAGE <name> IS
   <sichtbare Vereinbarungen>;
   [PRIVATE <Vereinbarungen>;]
END <name>

PACKAGE BODY <name> IS
   <lockale Vereinbarungen>;
BEGIN
   <Anweisungen>
   [EXCEPTION <Ausnahmebehandlung>]
END <name>;
```

Aufteilung von Spezifikation und Rumpf

Die Spezifikation muss vor dem Rumpf übersetzt werden. Ein fehlender Rumpf kann durch Stummelrumpf (stub unit) ersetzt werden.
Beispiel:

```
PACKAGE BODY master IS
...
   PROCEDURE q (x,y:REAL) IS
   SEPARATE;
...
```

Der eigentliche Rumpf kann dann unter Angabe des Pakets übersetzt werden:

```
SEPARATE (master)
   PROCEDURE q (x,y:REAL) IS
   BEGIN
   ...
   END q;
```

Private Typen

- Realisierung privater Typen kann zwar gesehen werden (z.B. wegen Speicherbedarf), aber nicht geändert werden
- Typ im Rumpf muss mit Spezifikation im Kopf übereinstimmen
- Eine Änderung der Spezifikation erfordert die Neuübersetzung aller abhängigen Bibliothekseinheiten bzw. benutzenden Programme

Syntax:

- in Spezifikation eines Paketes: TYPE <name> IS [LIMITED] PRIVATE
- im Rumpf: PRIVATE <Spezifikation von name>
- ohne LIMITED: ausserhalb muss Zuweisung und Vergleich für diesen Typ existieren
- mit LIMITED: Zuweisung und Vergleich durch eigene Paketfunktionen

Beispiel: Spezifikation des Pakets

```
PACKAGE complex_numbers IS
   TYPE compl IS PRIVATE;
   i: CONSTANT compl;
   FUNCTION + (x,y: compl) RETURN compl;
   FUNCTION - (x,y: compl) RETURN compl;
   PRIVATE
      TYPE compl IS
         RECORD
            re,im: FLOAT;
         END_RECORD;
      i: CONSTANT compl := (0.0,1.0);
   END complex_numbers;
```

Beispiel: Body

```
PACKAGE BODY complex_number IS
...
   FUNCTION + (x,y: compl) RETURN compl IS
   BEGIN
      ...
   END +

   FUNCTION - (x,y: compl) RETURN compl IS
   BEGIN
      ...
   END -
END complex_numbers;
```

Beispiel: Benutzen des Pakets

```
DECLARE
WITH complex_numbers;
USE complex_numbers;
u,v: compl;

BEGIN
   u:=(2.5,1.0);
   v:=i;
   u:=u+v;
END
```

Erläuterungen

- WITH <Paket>
  - Benutzung eines Paketes bei der Übersetzung
  - Paket muss bei der Übersetzung schon bekannt sein (mindestens mit ihrem Definitionsteil)
    $\Rightarrow$ zyklische Relationen sind nicht möglich
- USE <Paket>
  - im Rumpf kann der qualifizierende Name entfallen

Generische Einheiten

Generische Einheiten

- Unterprogramme oder Pakete
- dienen als Programmschablonen zur Mehrfachverwendung von Algorithmen
- freie Parameter (generic parameters) nach dem Wortsymbol GENERIC
- Parameter sind Objekte und Objekttypen
- freie Parameter werden bei der Übersetzung der Ausprägung durch aktuelle Parameter ersetzt (entspricht templates in C++)
- Ausprägung wird durch NEW instantiiert

Beispiel: Ringpuffer (nur Spezifikation)

```
GENERIC
   TYPE sometype IS PRIVATE;
PACKAGE queue_handling IS
   TYPE queue (maxlength: NATURAL)
   IS PRIVATE;

   PROCEDURE enqueue (q: IN OUT queue; elem: IN sometype);
   ...
   PROCEDURE dequeue ...
   FUNCTION nonempty ...
   FUNCTION nonfull ...
   PRIVATE
      SUBTYPE index
         IS CARDINAL RANGE 0..1000;
      TYPE queue (maxlength: NATURAL) IS
```

Beispiel: Ausprägung

```
DECLARE
   PACKAGE int_queue
      IS NEW queue_handling (INTEGER);
      USE int_queue;
      longqueue: queue (200);
      shortqueue: queue (10);
      BEGIN
         IF nonfull(shortqueue) THEN ...;
         ...
   END int_queue;
```

Objektorientierung

Objektorientierung

- Objektorientierung wurde in Ada 95 eingeführt
- basiert auf der Typerweiterung von RECORD (TAGGED)
- Beispiel:

```
TYPE messwert IS TAGGED
RECORD
   m1,m2: INTEGER;
END RECORD;
```

- Erweiterung um Zeitstempel

```
TYPE zeitmesswert
   IS NEW messwert WITH
   RECORD
      ankunft: time;
   END RECORD;
```

Objektorientierung

- abgeleitete Typen erben die Operationen des Vorgängers, können diese umdefinieren und neue hinzufügen
- in Ada95 trägt die gesamte Ableitungshierarchie (Grundtyp mit Ableitungen) die Bezeichnung „Klasse“
- Zum Grundtyp T existiert der Klassentyp T'Class (polymorpher Typ), der als formaler Parameter beim Überladen von Unterprogrammnamen dienen kann
- beim Aufruf mit einem aktuellen Parameter aus der Ableitungshierarchie wird dynamisch die passende Operation verwendet
- dieser Vorgang heisst in Ada95 dispatching (spätes Binden)

Abstrakte Typen

Ada95 unterstützt auch abstrakte Typen:

- auf obersten Niveau können abstrakte Typen (d.h. ohne Angabe des Inhalts und abstrakte Unterprogramme (ohne Angabne eines Rumpfes) definiert werden
- in der Ableitugshierarchie kann dieser Grundstock von allgemeinen Typen und Operationen auf verschiedene Weise realisiert werden
- der Anwender sucht sich unter den Varianten, die für seine Aufgabe geeignete Implementierung aus
- Beispiel:
  - abstrakter Typ: Menge
  - abstrakte Operationen: einfügen, löschen, suchen, vereinigen
  - die Menge kann auf unterschiedliche Art und Weise implementiert werden: z.B. Felder, Listen

Hierarchiscke Bibliotheken

Problem: Bei Wiederverwendung vorhandener Pakerte sollen Erweiterungen nicht zur Neuübersetzung aller bisherigen Benutzers des Grundpaketes führen (da sie die neuen Operationen nicht verwenden).
$\Rightarrow$ Einführung von Kindpaketen (child packages) und von hierarchischen Paket-Strukturen (hierarchical libraries)

- Ein Kund erbt vom Eltern-Paket alle sichtbaren Operationen und darf im Rumpf alle privaten Spezifikationen des Elternpakets verwenden
- Durch Weitervererbung wird eine Paket-Hierarchie aufgebaut
- Das Kind-Paket enthält bei Spezifikation den (qualifizierenden) Namen des Elternpakets

Beispiel: Vater-Paket Komplexe Zahlen

```
PACKAGE complex_numbers IS
   TYPE compl IS PRIVATE;
   FUNCTION + (x, y: compl) RETURN compl;
   PRIVATE
      TYPE compl IS
      RECORD
         re, im: FLOAT;
      END RECORD;
END complex numbers;
```

Beispiel: Kind-Paket (Erweiterung um polare Koordinaten)

```
PACKAGE complex_numbers.polar IS
   TYPE polar IS PRIVATE;
   FUNCTION abs (x: compl) RETURN FLOAT;
   FUNCTION arc (x: compl) RETURN FLOAT;
   FUNCTION polar2compl (r, theta: FLOAT)
      RETURN compl;
   PRIVATE
      TYPE polar IS
      RECORD
         r, theta: FLOAT;
      END RECORD;
END complex numbers.polar;
```

Beziehungen

- Kindpakete können als PRIVATE spezifiziert werden
- Ada95 definiert Regeln, die die Beziehung von Eltern-Kind und Geschwister-Paketen beschreiben, um zu verhindern, dass private Spezifikationen sichtbar gemacht werden können.
- Beispiel zu den Regeln:
  - Kontextumgebung des Elternpakets gilt auch für Nachkommen
  - WITH: Benutzung eines Kindpakets umfasst auch WITH für alle Vorfahren
  - ein Kind benötigt keine WITH Beziehung zu seinen Eltern jedoch zu den Geschwistern
  - ein sichtbares Kund kann kein PRIVATE Geschwisterpaket verwenden
  - Der Private.Teil und Rumpf jedes Kindes kann auch den PRIVAT.Teil seiner Eltern (und aller Vorfahren) benutzen.

Prozesse

Prozesse in Ada

- Für Prozesse wird der Datentyp TASK eingeführt.
- Tasks können auch Komponenten von Feldern oder Records darstellen.
- Tasks sind auch als Parameter erlaubt
- Tasks bekommen keine Parameter beim Start.
- Die Trennung in Spezifikationsteil und Rumpf ist obligatorisch
- Der Spezifikationsteil darf nur Eingänge (ENTRY für Aufrufe von Rendezvous-Anweisungen aus anderen Tasks deklarieren mit Angabe der formalen Parameter der zugehörigen ACCEPT-Anweisung im Rumpf

Syntax

```
TASK [TYPE] name IS
   ENTRY ename (<formale Parameter> );
   ENTRY ...
END name;

TASK BODY name IS
   <deklarationen>
BEGIN
   <Anweisungen>
   [EXCEPTION [<exception handler> ]]
END name;
```

bei Anweisungen u.a.:

```
ACCEPT ename (<formale Parameter> ) DO
. . .
END ename;
```

Lebenszyklus eines Prozesses

- Prozesse werden automatisch beim Abarbeiten der Deklaration aktiv, aber erst am Ende des Deklarationsteils gestartet
- Es gibt nur die Operation ABORT zum Datentyp TASK (gewaltsames Beenden)
- Durch die Blockstruktur können Prozessaufrufe geschachtelt auftreten.
- Prozesse terminieren automatisch beim Erreichen des Blockendes, falls sie nicht auf das Ende von untergeordneten Prozesse warten müssen.
- Der umfassende Prozess wird durch implizite Synchronisation des Betriebssysteme erst beendet, wenn alle in ihm deklarierten und damit alle gestarteten Prozesse beendet sind.
- Ein Block wird erst verlassen, wenn alle in ihm vereinbarten Prozesse beendet sind.

Partitionen

Partitionen

Seit Ada 95 kann ein Programm auch in mehrere Partitionen unterteilt werden.
Partition:

- haben einen eigenen Adressraum
- können Prozesse enthalten
- die Programme können durch Partitionen auf verschiedenen Rechnern ausgeführt werden
- aktive Partitionen enthalten Prozesse und main()
- passive Partitionen enthalten nur Daten und/oder Unterprogramme
- eine Partition wird erst beendet, wenn all ihre Prozesse beendet sind
- werden von aussen oder durch einen sogenannten Environment-Task angestoßen, bei deren Abarbeitung, die in ihr enthaltene Main-Prozedur aufgerufen wird
- Partitionen können mit anderen mittels RPC oder über gemeinsame Daten einer dritten Partition kommunizieren

Synchronisation

Synchronisation:

Ada bietet mit Rendezvous ein Konzept zur synchronen Kommunikation:

- Definition eines Eingangs in einem Prozess
- ACCEPT-Anweisung zu den ENTRYs in den Prozessen
- Der Aufruf des Eingangs eines anderen Prozesses erfolgt wie ein Prozeduraufruf mit Parametern
- sowohl der aufrufende als auch der aufgerufene Prozess warten bis die Anweisungen im ACCEPT durchgeführt sind
- alternatives Warten durch SELECT mit Guards (WHEN)
- Eine zeitliche Begrenzung der Wartezeit (watchdog) ist möglich.

Beispiel: Gemeinsamer Speicher

```
GENERIC
   TYPE item IS PRIVATE

PACKAGE readerwriter IS
   PROCEDURE read(x: OUT item);
   PROCEDURE write(x: IN item);
END;

PACKAGE BODY readerwriter IS
   V: item;

   TASK control IS
      ENTRY start;
      ENTRY stop;
      ENTRY write(x:item);
   END;
```

Beispiel: Verwaltung

```
TASK BODY control IS
   readers: integer :=0;
BEGIN
   ACCEPT write(x:IN item) DO
      v := x;
   END;
   LOOP
      SELECT
         WHEN write'count=0 =>
         ACCEPT start;
         readers:=readers+1;
      OR
         ACCEPT stop;
         readers:=readers-1;
      OR
         WHEN readers=0 =>
         ACCEPT write(x:IN item) DO
         v := x;
         END;
      END SELECT;
   END LOOP;
END control;
```

Beispiel: Lesen und Schreiben

```
PROCEDURE read(x:OUT item) IS
   BEGIN
      control.start; x:=v; control.stop;
   END read;

PROCEDURE write(x:IN item) IS
   BEGIN
      control.write(x);
   END write;

END readerwriter
```

Beispiel für Zeitbedingungen im Rendezvous:

```
SELECT
    ACCEPT this (...) DO
    ... END;
OR
    ACCEPT that (...) DO
    ... END;
OR
    DELAY 10.0;
    ... --time out statements
END SELECT;
```

Wechselseitiger Ausschluss

In Ada95 bietet zum wechselseitigen Ausschluss Geschützte Bereiche, sogenannte PROTECTED TYPES:

- Die Objekte können Typen und Daten sowie die benötigten Operationen (Funktionen, Prozeduren, ENTRY) enthalten
- Das Laufzeitsystem sichert, dass PROTECTED PROCEDURES exklusiv ausgeführt werden
- auf PROTECED FUNCTION kann gleichzeitig mehrfach lesend zugegriffen werden (nur IN-Parameter), da read-only Eigenschaft ohne Nebeneffekte
- Vorgehen ähnelt Monitoren
- Prioritätsvererbung ist möglich
- beim Auftreten von Ausnahmen wird der Block verlassen und die Belegung automatisch aufgehoben

Beispiel: Implementierung eines Semaphors

```
PROTECTED TYPE sema (init s: INTEGER := 1) IS
   ENTRY P;

   PROCEDURE V;

   PRIVATE
      count: INTEGER := init s;
END sema;
```

Beispiel

```
PROTECTED BODY sema IS
   ENTRY P WHEN count > 0 IS
   BEGIN
      count := count - 1;
   END P;

   PROCEDURE V IS
   BEGIN
      count := count + 1;
   END V;
END sema;
```

Beispiel: Aufruf

```
s : sema;
...
s.P;
... -- Exklusive Anweisungen
s.V;
```

Ausnahmen

Ausnahmen

- Ausnahmen können in Anweisungen, bei Deklarationen und im Rendezvous auftreten
- Der Benutzer kann Ausnahmen selbst definieren: ausn3: EXCEPTION
- vor- oder benutzerdefinierte Ausnahmen können durch RAISE ausgelöst werden
- Anweisungen zur Behandlung sind am Ende eines Rahmens anzugeben.
- Beim Auftreten einer Ausnahme wird der Rahmen verlassen und die Behandlung gestartet.
- Ist keine Behandlung angegeben, so wird die Ausnahme an den umgebenden Rahmen weitergeleitet (exception propagation), bis eine Behandlung oder ein Programmabbruch erfolgt

Ausnahmen

- Syntax der Behandlung

```
EXCEPTION
    WHEN exceptionname =>
        <Anweisungsfolge> ;
    ...
    WHEN OTHERS =>
        <Anweisungsfolge> ;
```

- mit OTHERS können beliebige Ausnahmen behandelt werden
- es gibt viele vordefinierte Ausnahmen, wie
  - constraint_error
  - numeric_error
  - programm_ error
  - storage_error
  - tasking_error

Unterbrechungen

Unterbrechungsbehandlung

- Verwendung von PROTECTED PROCEDURES als Antwortprogramme
- Die entsprechenden Prozeduren werden beim Auftreten eines Interrupts exklusiv mit genügend hoher Priorität ausgeführt
- Prioritätsbereich hängt vom Betriebssystem ab
- Die Prioritäten für Prozesse umfassen mindestens 30 Werte.
- Interrupts besitzen höhere Prioritätsklassen
- Die Zuordnung zu den Unterbrechungen erfolgt statisch oder dynamisch
- Identifikatoren der Unterbrechungen vom Herrsteller im Paket Ada.Interrupt.Names

Beispiel statische Zuordnung

```
PROTECTED PROCEDURE alarm IS
   PROCEDURE response;
      -- muss parameterlos sein
   PRAGMA
   attach handler (response, Alarm Id);
END alarm;

PROTECTED PROCEDURE BODY alarm IS
   PROCEDURE response IS
   ...
   END response;
END alarm;
```

Beispiel dynamische Zuordnung

```
PROTECTED PROCEDURE alarm IS
   PROCEDURE response;
      --muss parameterlos sein
      PRAGMA interrupt handler (response);
   END alarm;

PROTECTED PROCEDURE BODY alarm IS
   -- wie oben

   -- spaeterer Prozeduraufruf:
   attach handler (alarm.response, Alarm Id);
```

Real-Time Java

Java

- Sehr weit verbreitete Programmiersprache
- Vorteile:
  - Portabler Code durch Virtuelle Maschine
  - Objektorientiert
  - Garbage Collection
  - strengere Typisierung

Nachteile

Java hat vor allem in Hinblick auf Echtzeitsysteme diverse Nachteile:

- langsamer als beispielsweise C/C++
- zeitliche Vorhersagen können schwer getroffen werden (z.B. wegen Garbage Collection)
- Interaktion mit Hardware wird nicht direkt unterstützt
- Umgang mit Prioritäten schwierig
- keine klare Politik bei Monitoren

Ansatz: Real-Time Java

- Mehrere Gruppen arbeiten an Versionen für Real-Time Java
- Wichtigste Gruppe: Real-Time for Java Expert Group (RTJEG)
- Erste endgültige Version Dezember 2001
- PDF-Dokument mit Syntax und Semantik unter http://www.rtj.org/

Wesentliche Konzepte:

- präemptiver Scheduler mit mindestens 28 Prioritäten
- FIFO Scheduling und Prioritätsvererbung
- Einführung von Threads, die den Heap nicht benutzen (NoHeapRealTimeThread) und deshalb auch nicht von der Garbage Collection betroffen sind
- Einführung von speziellen Warteschlangen zur Kommunikation zwischen Threads
- Unterscheidung von verschiedenen Speicherarten (physical, immortal, scoped)
- Unterstützung von direktem Speicherzugriff
- Unterstützung von asynchroner Ausnahmen und Unterbrechungsbehandlung

Vorlesung Echtzeitsysteme - Uhren

Prof. Alois Knoll

Lehrstuhl Embedded Systems and Robotics

Sommersemester 2005

Inhalt

- Motivation
- Uhren
- Synchronisation

Literatur

- Leslie Lamport, Synchronizing clocks in the presence of faults, 1985

Links:

- http://www.ptb.de/
- http://www.maa.mhn.de/Scholar/dt_times.html
- http://www.ntp.org/

Definition Zeit

- Jeden Tag gegen Mittag erreicht die Sonne ihren höchsten Punkt am Himmel.
- Die Zeitspanne zwischen zwei aufeinanderfolgenden Ereignissen dieses Typs heisst Tag (genauer gesagt: ein Sonnentag).
- Eine Sonnensekunde ist 1/86400 dieser Spanne.
- Die Zeitmessung findet heute etwas einfacher statt, mit Hilfe von Atomuhren: eine Sekunde ist die Zeit, die ein Cäsium-133-Atom benötigt, um 9.192.631.770 mal zu schwingen.
- Am 1.1.1958 entsprach diese Atomsekunde genau einer Sonnensekunde (wegen der Erdreibung wird die Sonnensekunde immer länger).

TAI (Temps Atomique International)

- Atomzeitskala, die zur Koordination nationaler Atomzeiten ermittelt wird.
- An der Berechnung tragen 50 verschiedene Zeitinstitute mit derzeit etwa 250 Atomuhren bei.
- Zeit basiert auf der Atomsekunde.
- Der Referenzzeitpunkt ist der 1.Januar 1970
- relativen Genauigkeit von $+/-2*10^{-14}$ (Stand von 1990)
- aber keine exakte Übereinstimmung mit der Sonnenzeit

UTC (Coordinated Universal Time)

- eigentlicher Nachfolger der Greenwitchzeit
- realisiert durch Atomuhren, die Zeiteinheit ist die SI-Sekunde $\Rightarrow$ hochkonstante Zeiteinheit
- zusätzlich Übereinstimmung mit dem Sonnenlauf $\Rightarrow$ einheitliche Grundlage für die Zeitbestimmung im täglichen Leben
- durch Einfügen von Schaltsekunden wird UCT mit der universellen Sonnenzeit (UT1) synchronisiert
- Anpassung erfolgt zumeist zu Ende oder Mitte des Jahres

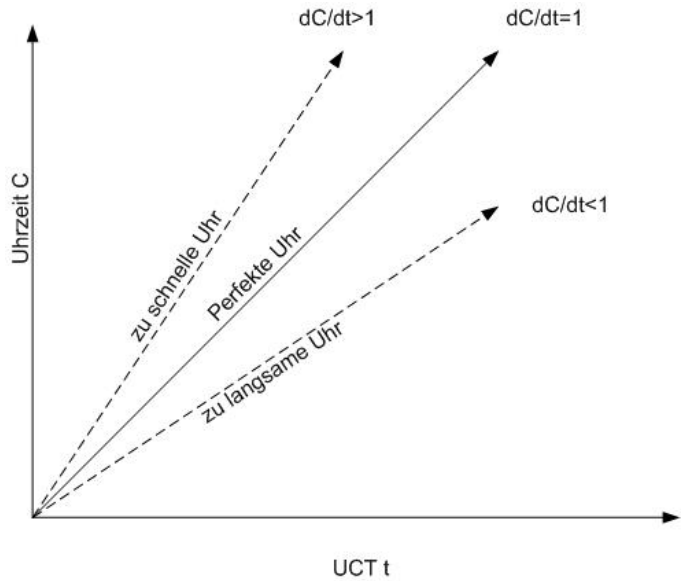
Uhren

Aufgaben

- Absolutzeitgeber
  - Datum, Uhrzeit
  - zeitabhängige Aufträge
  - Zeitstempel, Logbuch
  - Ursache-Wirkung-Feststellung
- Relativzeitgeber
  - verzögern
  - Messen von Zeitabständen
  - zyklisch anstoßen, abtasten
  - Zeitüberwachung von Wartezuständen

Genauigkeit von Uhren

- Der Wert der Uhr zum UTC Zeitpunkt t is $C(t)$
- Chips haben eine Genauigkeit von ca. 10^{-5}
- Bei einer Zeitdauer von 60 Stunden sollte eine Uhr 216.000 Mal pro Stunde ticken
- Realistisch ist ein Wert zwischen 215.998 und 216.002
- Eine Uhr arbeitet korrekt, wenn sie die vom Hersteller angegebene maximale Driftrate τ einhält, auch wenn sie dann etwas zu schnell oder zu langsam ist.



Verhalten einer Uhr

korrekt:

1. innerhalb der zugesichterte Gangabweichung

inkorrekt:

2. Verlassen der zugesicherten Gangabweichung
3. Zustandsfehler (z.B. Sprung im Zählerwert)
4. Stehenbleiben der Uhr

unmöglich

5. rückwärtslaufende Uhr
6. unendlich schnelle Uhr

Achtung: Auch zwei korrekte Uhren können sich unendlich weit voneinander entfernen, wenn sie nicht synchronisiert werden.

Synchronisation

Annahmen zur Synchronisation

1. Alle Uhren besitzen zu Beginn in etwa die gleiche Zeit.
2. Die Uhren fehlerfreier Prozesse gehen annähern richtig, d.h. sie besitzen gemeinsame Ganggenauigkeit.
3. Ein fehlerfreier Prozess p kann die Differenz seiner Uhr und der von Prozess q mit einer Genauigkeit ϵ lesen.

Anforderungen

1. Zu jedem Zeitpunkt zeigen die Uhren zweier fehlerfreier Prozesse ungefähr den gleichen Wert.
2. Durch die Synchronisation entstehen keine (bzw. nur sehr kleine Zeitsprünge)
3. Insbesondere darf die Kausalität nicht verletzt werden (z.B. Zurückstellen der Zeit)

Andernfalls kann eine konsistente Ausführung (z.B. wegen Anweisungen mit absoluten Zeitangaben) nicht garantiert werden.

Unterscheidung der Verfahren

Die Synchronisation erfolgt in der Regel periodisch.
Man unterscheidet die Verfahren nach:

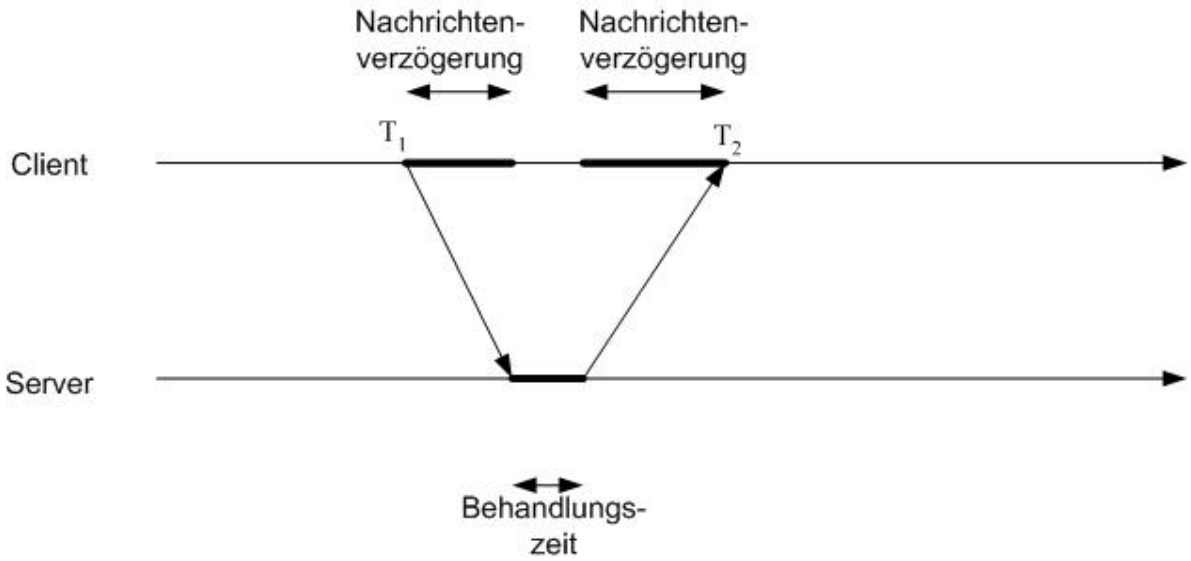
1. externer Synchronisation: die Synchronisation erfolgt anhand einer externen, als perfekt angenommenen Uhr
2. interner Synchronisation: die Uhren einigen sich gemeinsam auf eine korrekte Zeitbasis
3. zentraler Synchronisation: eine Einheit koordiniert die Synchronisation $\Rightarrow$ fehleranfällig
4. verteilter Synchronisation: alle Einheiten beteiligt $\Rightarrow$ hohes Datenaufkommen

Bei der externen Synchronisation muss der maximal tolerierte Fehler nur halb so groß sein, wie der Fehler bei interner Synchronisation. Warum?

Algorithmus von Cristian (1989)

Das Verfahren basiert auf verteilter, externer Synchronisation.

- Innerhalb des Systems existiert ein Time-Server, der zumeist ein UTC-Empfänger darstellt.
- In regelmäßigen Abständen senden die anderen Einheiten einen Time-Request, der so schnell wie möglich vom Server beantwortet wird.



1.Problem:

Die lokale Uhr könnte nun auf die empfangene Zeit gesetzt werden.

Problem: Zeitsprünge würden entstehen.

Lösung: Die Uhr wird graduell angepasst. Zum Beispiel wird das Interval zwischen zwei Uhrenticks von 1ms auf 0.9 ms heruntergesetzt, wenn die Uhr zu langsam war.

2.Problem:

Problem: Die Nachricht ist veraltet, wenn die Nachrichtenverzögerung und die Behandlungszeit nicht vernachlässigbar sind.

Lösungen:

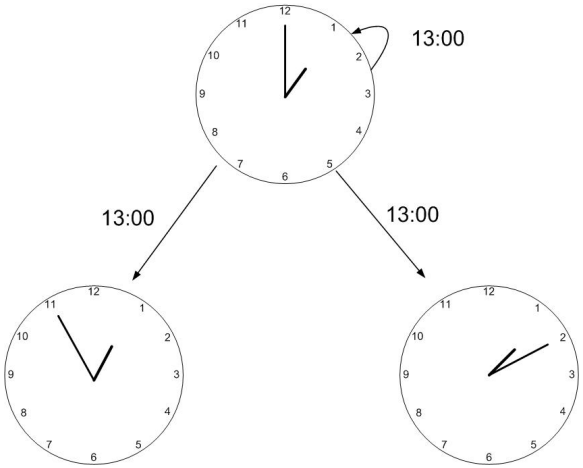
- Lösung von Cristian: Messung der Nachrichtenverzögerung
- Abschätzung, falls Informationen fehlen: $(T_1 - T_2)/2$
- Falls die Behandlungszeit bekannt ist, kann das Ergebnis weiter verbessert werden.
- Zusätzliche Verbesserung: Ermitteln eines Durchschnittwertes, Ausreisser müssen dabei ausser acht gelassen werden.

Algorithmus von Berkeley

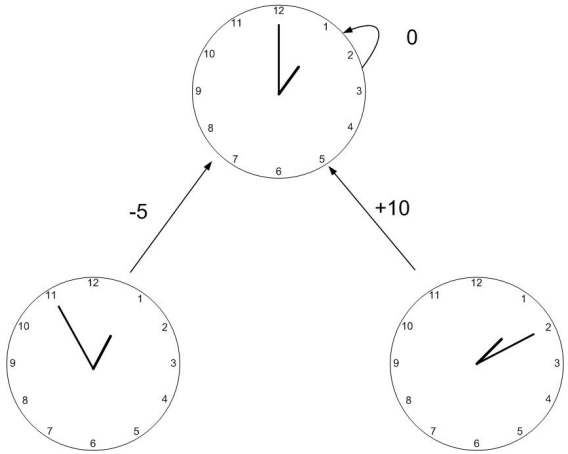
- Entwickelt 1989 in Berkeley
- Annahme: es steht kein UTC-Empfänger zur Verfügung
- Ein Rechner agiert als aktiver Time-Server.
- Der Server frägt die Zeiten aller anderen Rechner ab (Phase 1) und ermittelt den Durchschnittswert (Phase 2).
- Dieser wird als aktueller Uhrenwert (Phase 3) an alle anderen gegeben.

Zentrales, internes Verfahren.

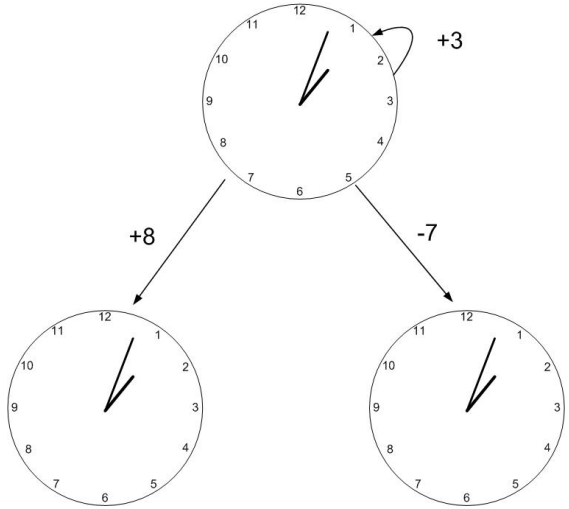
Beispiel



Phase 1



Phase2



Phase 3

Network Time Protocol

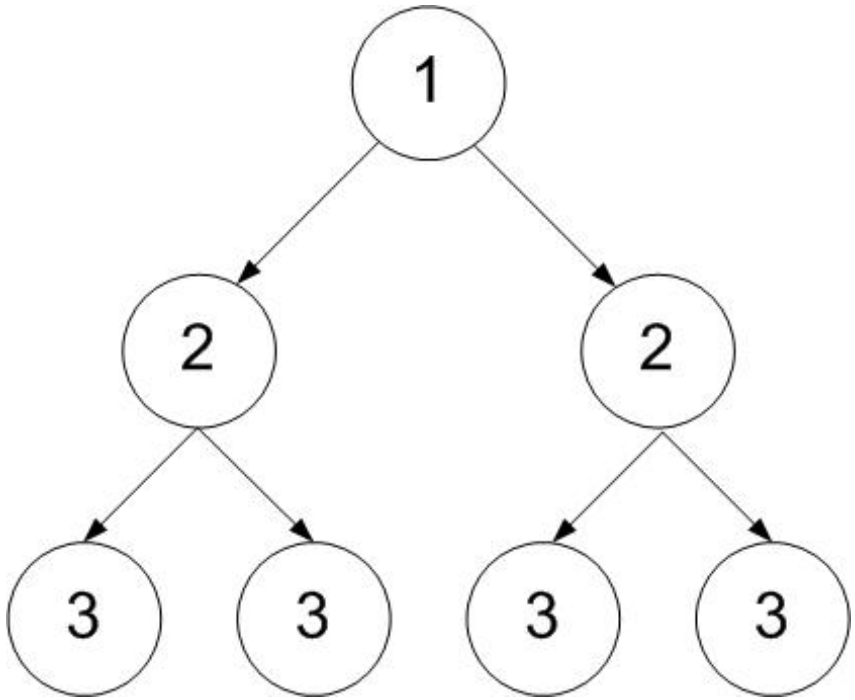
Problem: Die angegebenen Algorithmen funktionieren nur in kleinen statischen Netzen.

Das NTP Protokoll bietet eine Möglichkeit in großen Netzen eine Synchronisation zu gewährleisten. Die Netze können dabei dynamisch konfiguriert werden, um eine zuverlässige Synchronisation zu gewährleisten.

Die Grundstruktur von NTP ist ein hierarchisches Modell.

Struktur

- Der Dienst wird durch ein verteiltes Serversystem geleistet.
- Primäre Server sind direkt mit einer UTC-Quelle verbunden.
- Sekundäre Server synchronisieren sich mit primären Servern usw.



Funktionsweise

- Zur Erhöhung der Zuverlässigkeit ist das Servernetzwerk rekonfigurierbar.
- Anhand von Qualitätsmessungen (Uhrengenauigkeit, Nachrichtenverzögerungen) durch die Server können Clients die beste Quelle wählen.

Qualität: 99% aller NTP-Clients im Internet haben einen Synchronisationsfehler kleiner 30ms. Alle Rechner sind innerhalb von 50ms synchronisiert.

Synchronisation bei fehlerhaften Uhren

Problemstellung

Die bisherigen Algorithmen basierten alle auf der Annahme der fehlerfreien Uhren.

Im Folgenden werden Algorithmen betrachtet, die mit einer maximalen Anzahl von fehlerbehafteten Uhren m umgehen können. Insgesamt soll das System aus n Uhren bestehen. Betrachtet werden im Besonderen auch byzantinische Fehler (die fehlerfreie Einheit kann beliebige Ausgaben produzieren).

Die maximal zulässige Abweichung zweier Uhren bezeichnen wir mit ϵ.

In Frage kommen dabei nur verteilte Algorithmen, um einen Single-Point-of-Failure auszuschließen.

Konvergenzalgorithmus

Jede Einheit liest die Uhr der anderen Rechner und berechnet den Mittelwert. Ist die Abweichung einer Uhr größer als ϵ, so verwendet der Algorithmus stattdessen den Wert der eigenen Uhr.

Der Algorithmus arbeitet erfolgreich, falls gilt: $n \geq 3m$

Beweis

Seien p, q zwei fehlerfreie Einheiten, r eine beliebige Einheit.

Sei $t(p, r)$ die Uhrzeit, die die Einheit p für die Mittelwertsberechnung verwendet.

$\Rightarrow$ r fehlerfrei: $\Delta(p, r) \approx \Delta(q, r)$
$\Rightarrow$ r fehlerbehaftet $|\Delta(p, r) - \Delta(q, r)| \leq 3\epsilon$

Fortsetzung Beweis

Einheit p stellt seine Uhr auf: $\frac{1}{n}\sum_r t(p,r)$
Einheit q stellt seine Uhr auf: $\frac{1}{n}\sum_r t(q,r)$

Schlechtester Fall:

- $(n-m)$-mal fehlerfrei: $t(p,r) \approx t(q,r)$
- m-mal fehlerbehaftet $|\Delta(p,r) - \Delta(q,r)| \leq 3\epsilon$

$\Rightarrow$ Differenz beider Uhren
$p,q = \frac{1}{n}|\sum_r t(p,r) - \sum_r t(p,r)| \leq \frac{m}{n}3\epsilon < \epsilon$

Vorlesung Echtzeitsysteme - Echtzeitfähige Kommunikation

Prof. Alois Knoll

Lehrstuhl Embedded Systems and Robotics

Sommersemester 2005

Inhalt

- Grundlagen
- Medienzugriffsverfahren und Vertreter
  - Ethernet
  - CAN-Bus
  - TTP
  - Token Ring, FDDI

Literatur

- Andrew S. Tanenbaum, Computernetzwerke, 2003
- TTTech Computertechnik AG, Time Triggered Protocol TTP/C High-Level Specification Document, 2003

- http://www.can-cia.org/
- http://standards.ieee.org/getieee802/portfolio.html
- http://www.vmars.tuwien.ac.at/projects/ttp/
- http://www.tttech.com/

Anforderungen an Echtzeitfähige Kommunikation

Echtzeitsysteme stellen spezielle Anforderungen an die Kommunikation:

- vorhersagbare maximale Übertragungszeiten
- garantierte Bandbreiten
- effiziente Implementierungen
- Fehlertoleranz

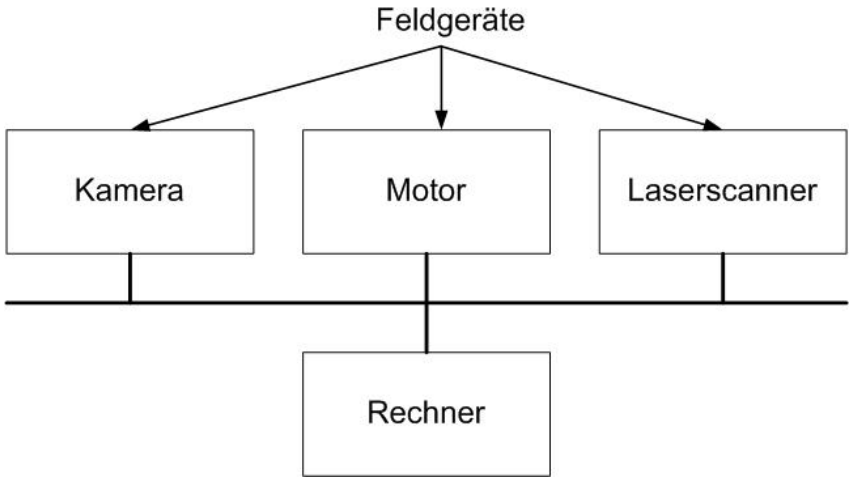
Kommunikationsmedien

Echtzeitsysteme operieren häufig unter extremen Bedingungen. Zudem sind Resourcen häufig sehr beschränkt und es besteht ein hoher Kostendruck. Die Entscheidung für ein Kommunikationsmedium basiert deshalb auf diversen Kriterien:

- maximale Übertragungsrate
- Dämpfung in db pro km
- Materialeigenschaften (z.B. für Installation)
- Störungsempfindlichkeit
- Kosten, Marktproduktpalette

Definition Feldbus

Kommunikationssysteme in Echtzeitanwendungen sind häufig in der Form von Feldbussen:

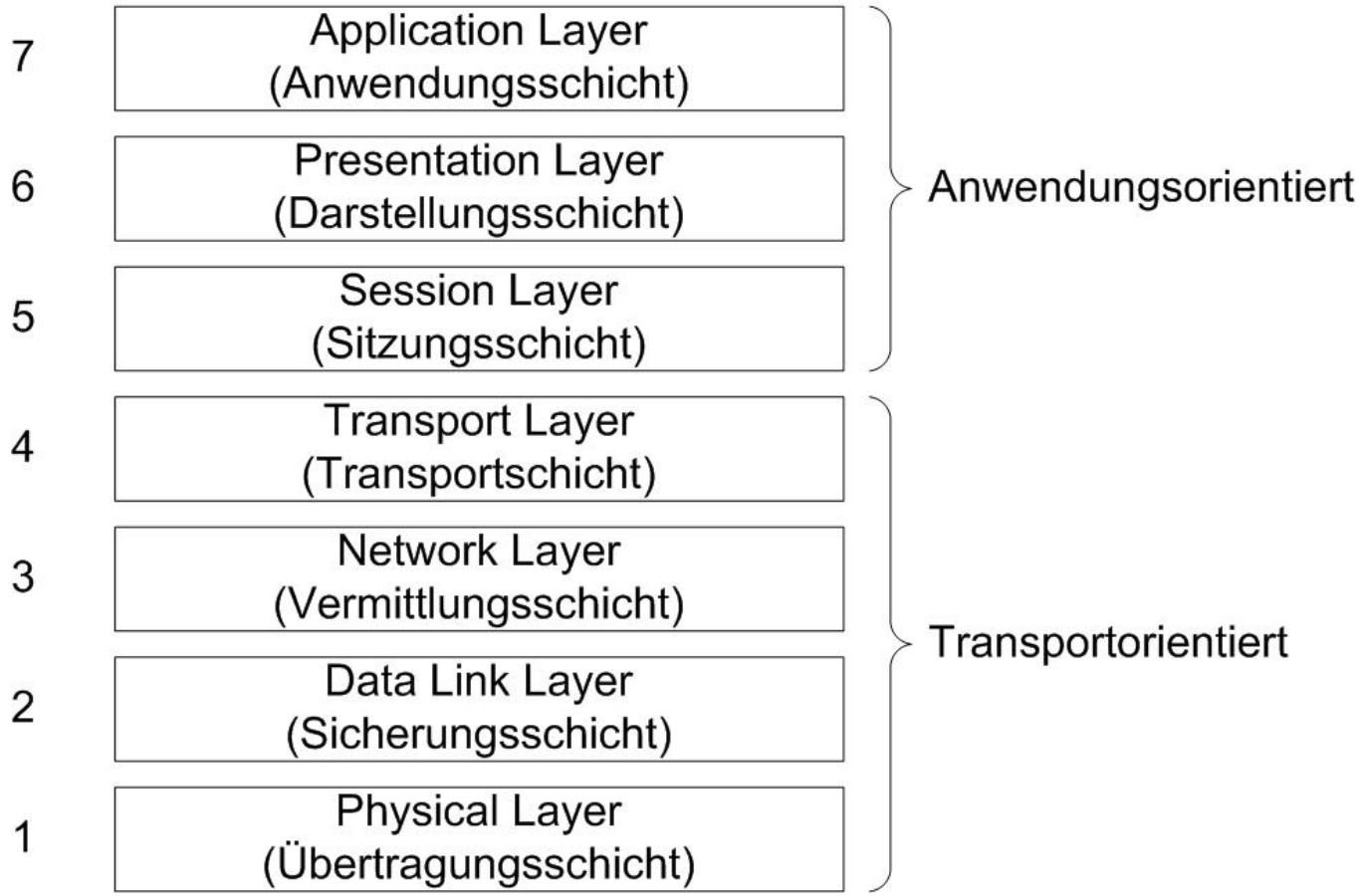


Als Feldgeräte werden Sensoren, Aktoren genannt. Auch Geräte, die eine Vorverarbeitung der Daten vornehmen werden zu den Feldgeräten gezählt.

Der Feldbus verbindet die Feldgeräte mit dem Steuerungsgerät.

Echtzeitkritische Nachrichten sind in der Regel kurz, unkritische Nachrichten länger.

ISO/OSI-Modell



Schicht 1

Aufgaben der **Übertragungsschicht:**

- Bitübertragung auf physikalischen Medium
- Festlegung der Medien
  - elektrische, optische Signale, Funk
  - Normung von Steckern
- Festlegung der Übertragungsverfahren/Codierung
  - Interpretation der Pegel
  - Festlegung der Datenrate

Schicht 2

Aufgaben der **Sicherungsschicht:**

- Gewährleistung einer weitgehend fehlerfreien Übertragung
  - Prüfsummen, Paritätsbits
  - Aufteilung der Nachrichten in Blöcke, Nummerierung der Blöcke
- Regelung des Medienzugriffs
- Flusskontrolle

Schicht 3

Aufgaben der **Vermittlungsschicht:**

- Aufbau von Verbindungen
- Weitervermitteln von Datenpaketen (Routing)
  - Routingtabellen
  - Flusskontrolle
  - Netzwerkadressen

Schicht 4

Aufgaben der **Transportschicht:**

- Transport zwischen Sender und Empfänger (End-zu-End-Kontrolle)
- Segmentierung von Datenpaketen
- Staukontroll (congestion control)

Schicht 5

Aufgaben der **Sitzungsschicht:**

- Auf- und Abbau zwischen Teilnehmern auf Anwendungsebene
- Einrichten von check points zum Schutz gegen Verbindungsverlust
- Bereitstellung von Diensten zum organisieren und synchronisiertem Datenaustausch
- Spezifikation von Sicherheitsmechanismen (z.B. Passwörtern)

Schicht 6

Aufgaben der **Darstellungsschicht:**

- Konvertierung der systemabhängigen Daten in unabhängige Form
- Datenkompression
- Verschlüsselung

Schicht 7

Aufgaben der **Anwendungsschicht:**

- Bereitstellung anwendungsspezifischer Übertragungs- und Kommunikationsdienste
- Beispiele:
  - Datenübertragung
  - E-Mail
  - Virtual Terminal
  - Remote login

Zeiten für Nachrichtenübertragung

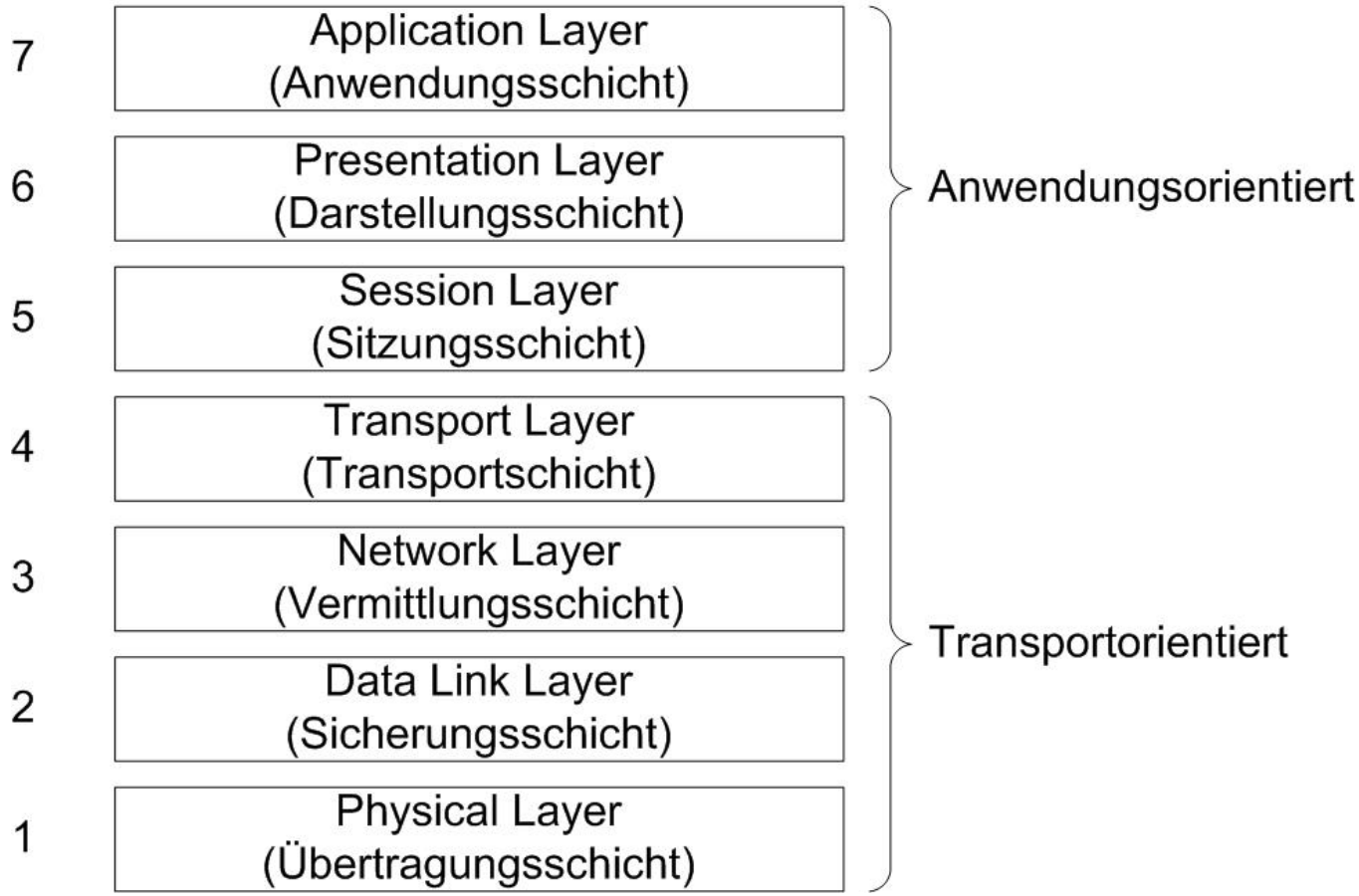
Die Zeit, die für das Versenden einer Nachricht benötigt wird, setzt sich aus unterschiedlichen Komponenten zusammen:

- Umsetzung der Protokolle der einzelnen Schichten durch den Sender
- Warten auf Medienzugang
- Übertragungszeit auf Medium
- Entpacken der Nachricht in den einzelnen Schichten durch den Empfänger

$\Rightarrow$ Mit jeder zu durchlaufenden Schicht verlängert sich die Übertragungszeit. Zudem vergrößert jede Schicht die zu sendenden Daten.

$\Rightarrow$ In Echtzeitsystemen wird die Anzahl der Schichten zumeist reduziert.

ISO/OSI-Modell



Schichten in Echtzeitsystemen

In Echtzeitsystemen werden typischerweise nur folgende Schichten implementiert:

- Anwendungsschicht
- Sicherungsschicht
- Physikalische Schicht

Zugriffsverfahren

Problemstellung

Zugriffsverfahren regeln die Vergabe des Kommunikationsmediums an die einzelnen Einheiten. Eine Überlagerung mehrerer Nachrichten kann zu einer Verfälschung der Nachrichten führen und muss deshalb zumindest erkannt werden.

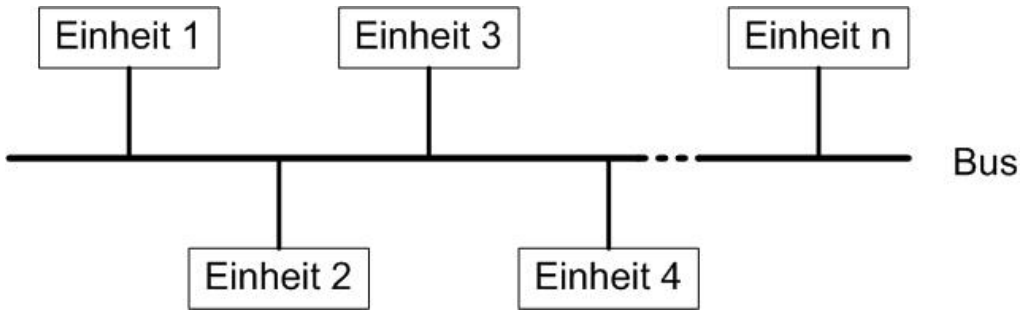
Zugriffsverfahren können dabei in verschiedene Klassen eingeteilt werden:

- Erkennung von Kollisionen (CSMA/CD)
- Vermeiden von Kollisionen (CSMA/CA)
- Verhindern von Kollisionen (token-basiert, TDMA)

CSMA/CD (Vertreter Ethernet)

CSMA

CSMA steht für **Carrier Sense Multiple Access**:
⇒ alle am Bus angeschlossenen Einheiten können die Daten lesen.
⇒ mehrere Einheiten dürfen Daten auf den Bus schreiben.



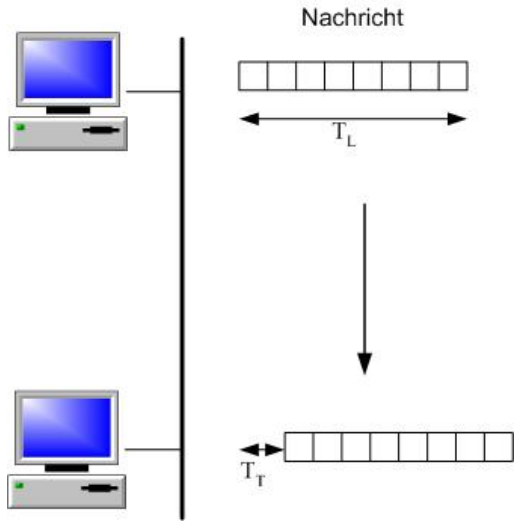
CSMA/CD

1-persistentes CSMA/CD (CD := Collision Detection) ist ein relativ simples Verfahren und in der Norm IEEE 802.3 spezifiziert.

Ablauf:

1. Test, ob Leitung frei ist (carrier sense)
2. Falls Leitung für die Zeitdauer eines **IFS (inter frame spacing)** frei ist, beginne Übertragung, ansonsten fahre fort mit 5.
3. Übertrage die Daten und kontrolliere die Leitung auf Kollisionen. Bei einer entdeckten Kollision: schicke Jam-Signal (damit alle anderen Sender die Kollision auch entdecken) und fahre mit Schritt 5 fort.
4. Übertragung erfolgreich beendet: Benachrichtige höhere Schicht
5. Warten bis Leitung frei ist
6. Sobald Leitung frei: weitere zufälliges Warten (z.B. Backoff-Verfahren) und Neustarten mit Schritt 1, falls maximale Sendeversuchsanzahl noch nicht erreicht.
7. Maximale Anzahl an Sendeversuchen erreicht: Fehlermeldung an höhere Schicht.

Transferzeit und Latenzzeit



$\Rightarrow$ Die Transferzeit T_T muss deutlich kleiner als die Latenzzeit T_L sein, um Kollisionen entdecken zu können.

Backoff-Verfahren in Ethernet

Würde nach einer Kollision nicht eine zufällige Zeit gewartet werden, käme es sofort zu einer erneuten Kollision.

Lösung in Ethernet: Die Konfliktparteien wählen eine zufällige Zahl d aus dem Interval $[0...2^i]$, wobei i die Anzahl der bisherigen Kollisionen ist. Mit ansteigendem i wird eine Kollision immer unwahrscheinlicher. Bei $i = 16$ wird die Übertragung abgebrochen und ein Systemfehler vermutet.

Beurteilung: Ethernet

Es können keine Garantien für Übertragungszeiten gegeben werden, deshalb ist Ethernet nicht für Echtzeitsysteme geeignet.

Verbesserungsansätze: Reduzierung der Kollisionen durch Aufteilung des Netzes in verschiedene Kollisionsdomänen (z.B. switched ethernet).

CSMA/CA (Vertreter CAN)

CSMA/CA

Wie bei Ethernet gesehen reicht eine reine Entdeckung von Kollisionen nicht aus, um eine echtzeitfähige und sichere Kommunikation zu garantieren. Vielmehr muss ein Mechanismus geschaffen werden, der es erlaubt eine Kollision schon vor der Korrumpierung der Daten zu entdecken und geeignet darauf zu reagieren.

$\Rightarrow$ CSMA/**CA (collision avoidance)**: Durch die Einführung von Prioritäten wird beim Beginn einer Kollision sichergestellt, dass die höherpriorisierte Nachricht gesendet werden kann.

CAN-Protokoll

- **CAN (Controller Area Network)** wurde 1981 von Intel und Bosch entwickelt.
- Einsatzbereich vor allem im Automobilbereich und im Haushaltsgerätebereich
- Datenübertragungsraten von bis zu 1 Mbit/s, Reichweite max. 1km
- Einsatz von priorisierten Nachrichten wird unterstützt
- Insgesamt werden die Schichten 1,2 und 7 des ISO/OSI-Modells implementiert

CAN: Schicht 1

Als **Busmedium** wird ein Twisted-Pair Kabel mit einem Wellenwiderstand von 108-132 Ω empfohlen. Der Vorteil der Zweidrahtleitung liegt vor allem in der Möglichkeit der differentiellen Übertragung.

Die Datenübertragungsrate und die maximale Kabellänge sind aufgrund der Bitsynchronität der Stationen miteinander verknüpft. Folgende Konfigurationen sind beispielsweise möglich:

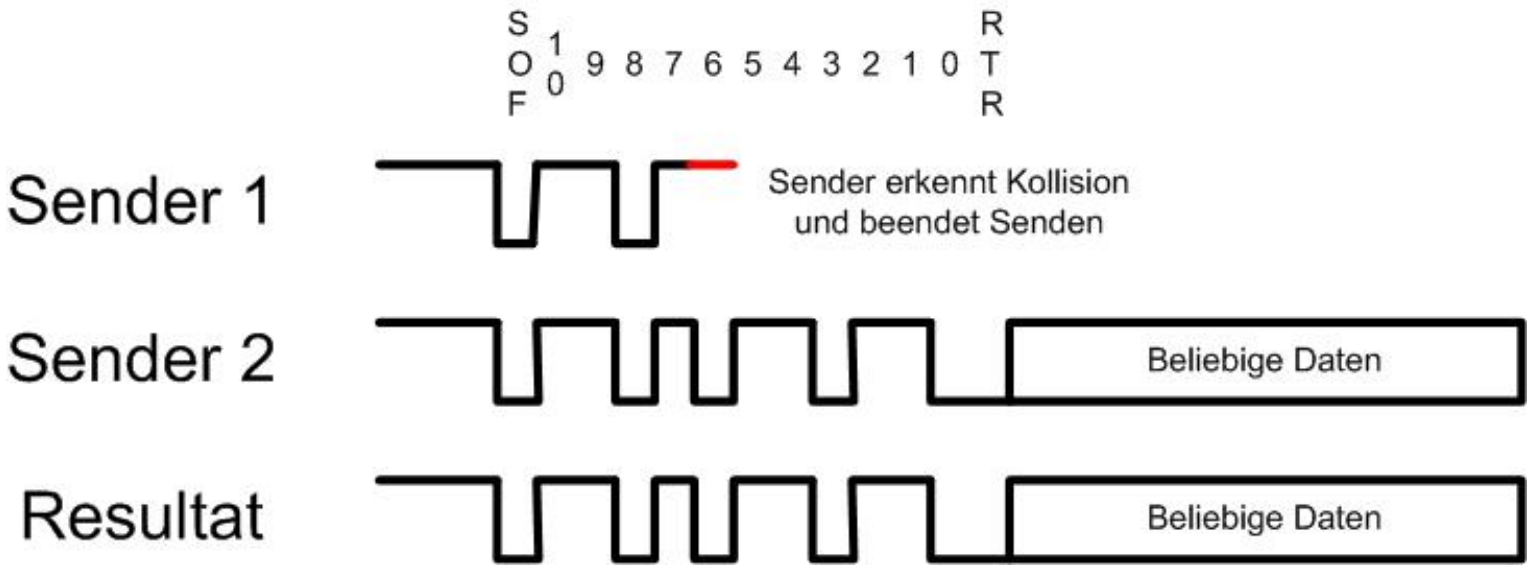
- 1 MBit/s, maximale Länge: 40m
- 500 kBit/s, maximale Länge: 100m
- 125 kBit/s, maximale Länge: 500m

CAN: Schicht 2

Das CAN-Protokoll spezifiziert für den Medienzugriff ein CSMA/CA-Verfahren:

- Die Übertragung der Bits erfolgt so, dass ein Bit je nach Wert **dominant** oder **rezessiv** wirkt. Dominante Bis überschreiben so rezessive Bits.
- Die Sender sind bitsynchronisiert.
- Jedem Nachrichtentyp (z.B. Sensorwert, Kontrollnachricht,...) wird ein Identifier zugewiesen.
- Es dürfen mehrere Empfänger Nachrichten mit dem gleichen Identifier zugewiesen bekommen
- Jeder Identifier darf nur einem Sender zugewiesen werden

Dominante Bits überschreiben rezessive Bits



CAN: Frametypen

- Datenframe: zum Versenden von maximal 64bit Daten.
- Remoteframe: zur Anforderung von Daten.
- Errorframe: dient zur Signalisierung einer erkannten Fehlerbedingung in der Übertragung an alle anderen Teilnehmer.
- Overloadframe: dient zur Zwangspause zwischen Remoteframe und Datenframe

CAN: Frame-Aufbau (Datenframe)

1	11	1	1	1	4	0...64	15	1	1	1	7	3
Start of frame	Identifier (Extended CAN 27bit)	Remote Transmission Bit	Identifier Extension Bit	reserviert	Datenlängenfeld	Datenfeld	CRC-Prüfsumme	CRC Delimeter	Bestätigungsslot	Bestätigungsdelimeter	End of Frame	Intermission

CAN: Frame-Aufbau

- **SOF**: dominantes Bit
- **Identifier**: Nachrichtentyp, entweder 11 oder 27 bit
- **RTR**: Remote Transmission Request. Gesetzt (dominant), falls Daten angefordert werden sollen.
- **CRC**: 15-Bit CRC-Prüfsumme plus 1 bit Delimiter (rezessiv)
- **ACK**: Bestätigungsfeld plus 1 bit Delimiter (rezessiv)
- **EOF**: End of Frame, 7 Bit (rezessiv)
- **Intermission**: mind. 3 rezessive Bits müssen zwischen 2 Nachrichten liegen

CAN: Schicht 7

- Im Gegensatz zu Schicht 1 und 2 existieren hier keine internationalen Normen
- Die Nutzerorganisation CiA e.V. versucht eine solche Norm CAL (CAN Application Layer) herbeizuführen
- Ziele:
  - Einheitliche Sprache zur Entwicklung von verteilten Anwendungen
  - Ermöglichung einer Interaktion von CAN-Modulen unterschiedlicher Hersteller

Probleme mit CSMA/CA

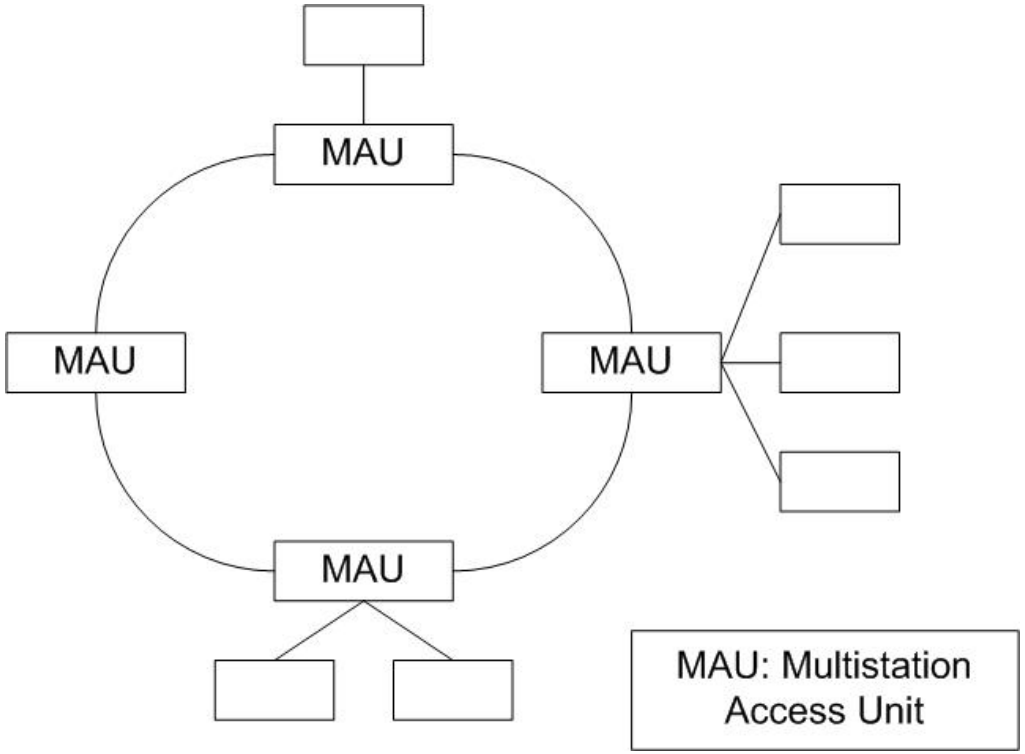
CSMA/CA-Verfahren besitzen einen großen Nachteil: die vorgeschriebene Bitsynchronität der Sender
$\Rightarrow$ Bandbreite und Länge des Netzwerkes sind begrenzt

Mögliche Lösungen: Tokenbasierte Verfahren

Tokenbasiere Verfahren (Vertreter: Token Ring)

Tokenbasierte Verfahren

Die Einheiten eines Rechners dürfen nur dann senden, wenn sie eine Berechtigung (Token) besitzen. Berechtigungen werden meistens zyklisch weitergegeben ⇒ **Token-Ring**.



Ein Token ist dabei eine spezielle Bitsequenz.

Token Ring

Token Ring, festgelegt im Standard IEEE 802.5, definiert Kabeltypen und Signalisierung für die Bitübertragungsschicht, Paketformate und Protokolle für die Medienzugriffskontrolle (Media Access Control, MAC)/Sicherungsschicht.

Das Verfahren erreicht Geschwindigkeiten von 4 und 16 MBit/s. Aufgrund der Kollisionsfreiheit ist dies mit den effektiven Übertragungsraten von 10 bzw. 100 MBit/s Ethernet vergleichbar. Als Codierung wird differentieller Manchester-Code ($\Rightarrow$ selbstsynchronisierend) verwendet.

Token Ring: Verfahren

- Die Station, die den Token besitzt, darf Rahmen auf dem Ring senden.
- Die Rahmen werden von Station zu Station übertragen.
- Die einzelnen Stationen empfangen die Daten und regenerieren sie zur Weitersendung an den Nachfolger.
- Empfänger der Nachricht kopieren die Nachricht zusätzlich und ändern das C-Bit auf 1(siehe Rahmen)
- Trifft der Rahmen wieder beim Sender ein, so entfernt in dieser aus dem Netz.
- Ein Sender darf den Token höchstens solange behalten, bis die Token-Haltezeit (Standard: 10ms) abgelaufen ist.

Token Ring: Prioritäten

- Jeder Station wird eine Priorität zugewiesen.
- Das Token besitzt ebenfalls einen 3-Bit Speicherplatz für die Priorität.
- Eine Station kann in den Prioritätsbits eines passierenden Datenrahmens die Priorität vormerken. Dabei darf die Priorität allerdings nur erhöht werden.
- Stationen dürfen Tokens nur dann annehmen, wenn ihre Priorität mindestens so hoch ist, wie die Priorität des Tokens

Token Ring: Monitor

Der **Monitor** sorgt für einen fehlerfreien Ablauf des Protokolls. Die Aufgaben sind:

- Entfernung von fehlerhaften Rahmen
- Entfernung von Rahmen, die schon das zweite Mal den Monitor passieren (aufgrund eines Fehlers im Empfänger) $\Rightarrow$ Überwachungsbit
- Einfügen eines Token bei Verlust
- Signalisierung der Existenz eines Monitors $\Rightarrow$ ACTIVE_MONITOR_PRESENT Rahmen, Standby Monitor Timer (TSM)
- Einfügen von Verzögerungen, so dass das Token immer genug „Platz“im Ring hat

Token Ring: Initialisierung

Bei der Initialisierung oder beim Ablauf des TSM werden folgende Schritte durchgeführt:

1. Sendung eines CLAIM_TOKEN Rahmens
2. Überprüfung, ob sonstiger Rahmen die Station passiert
3. Falls nein, wird die Station zum neuen Monitor
4. Generierung eines Tokens
5. Zusätzlich überprüft jede Station mittels DUPLICATE_ADDRESS_TEST Rahmen, ob die eigene Adresse vorhanden ist.

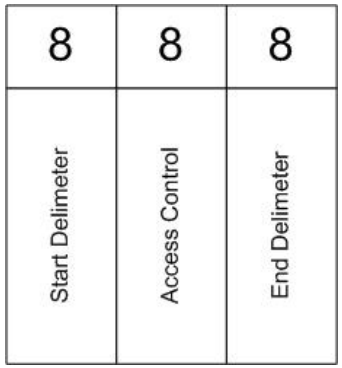
Token Ring: Rekonfigurierung

Ein Ausfall einer Station bedeutet eine Unterbrechung im Ring.

Mittels eines BEACON-Nachricht kann der Fehler erkannt werden.
Durch Überbrückungen kann der Ring wieder geschlossen werden.

Token Ring: Nachrichtenformate

Token-Ring unterscheidet zwei verschiedene Formate:



Token-Format

Frame-Format

Token Ring: Sequenzen

- **SD**: Codierung JK0JK000, wobei J und K jeweils die Manchester-Codierung verletzen (00 oder 11, kein Flankenwechsel in der Mitte des Taktes)
- **AC**: Codierung PPPTMRRR mit Priority-Bits P, Token Bit T, Monitor-Bit M und Priorität des nächsten Tokens R
- **FC**: Art des Rahmens (z.B. BEACON, CLAIM_TOKEN...)
- **DA,SA**: Adressen des Senders und des Empfängers (2 bzw. 6 Byte)
- **info**: zu übertragende Nachricht, Länge ist nicht begrenzt
- **FCS**: Prüfsumme von FC bis FCS
- **ED**: Codierung JK1JK1IE, wobei I anzeigt, dass weitere Rahmen folgen und E zur Signalisierung von Fehlern verwendet wird
- **FS**: Codierung ACrrACrr, wobei A anzeigt, dass der Empfänger den Rahmen erkannt hat und C, dass der Rahmen vom Empfänger kopiert wurde, r bezeichnet reservierte Bits

Vergleich Ethernet - Token Ring

Ethernet

+ weit verbreitet
+ Standardverkabelung
- Topologie (bei Bus) fehleranfällig (Separation)
- Endgeräte komplex
- keine deterministischen Zeitvorhersagen möglich
- keine Prioritäten
- bei hoher Last viele Kollisionen
+ bei niedriger Last schneller Medienzugriff

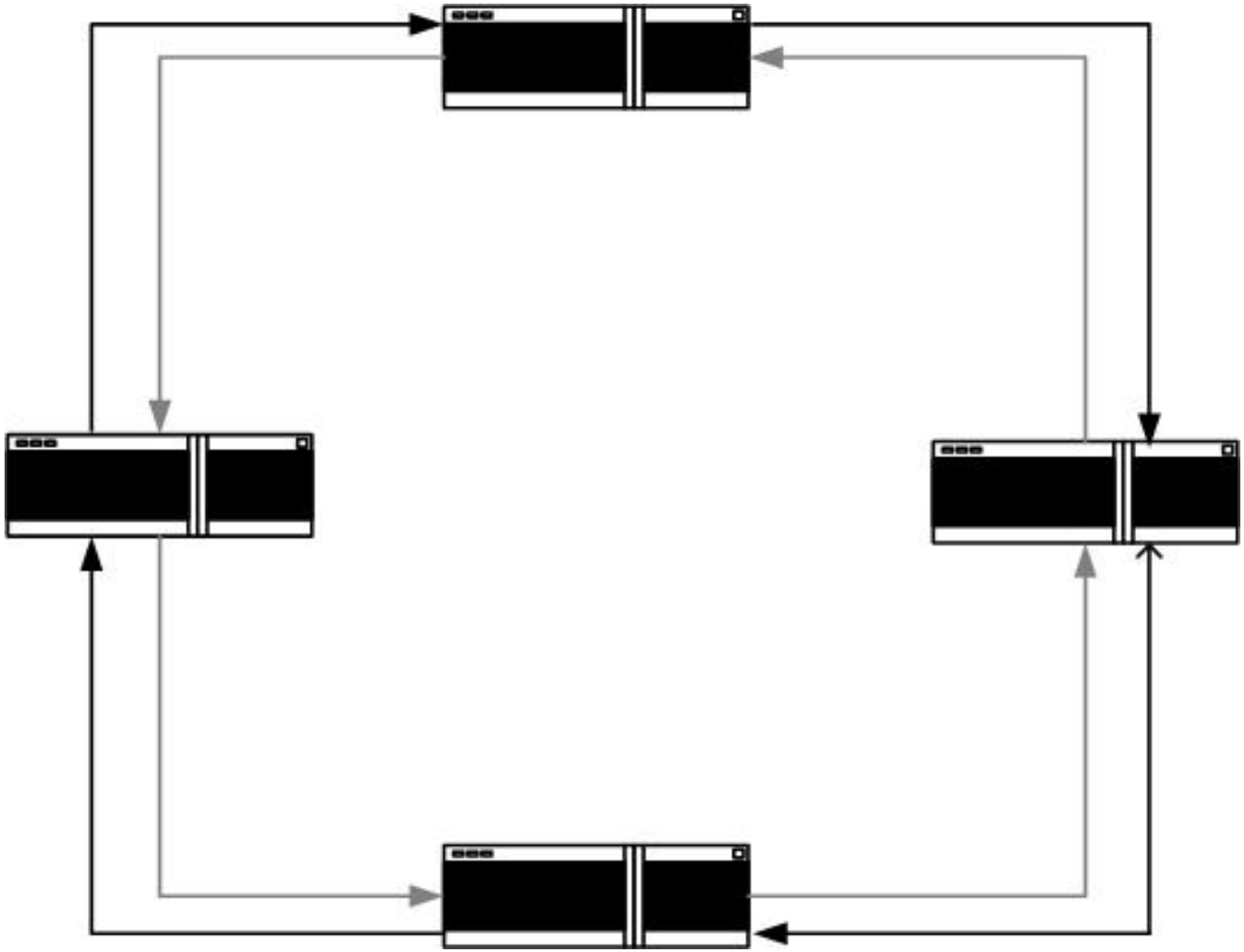
Token Ring

- Verbreitung geringer
o Topologie fehlertolerant (jedoch Verzögerungen)
+ Standardverkabelung
+ Endgeräte sehr einfach
+ deterministische Zeitvorhersagen
+ Prioritäten werden unterstützt
- Kollisionen ausgeschlossen
+ Auslastung optimal selbst bei hoher Last
- bei niedriger Last verzögerter Medienzugriff

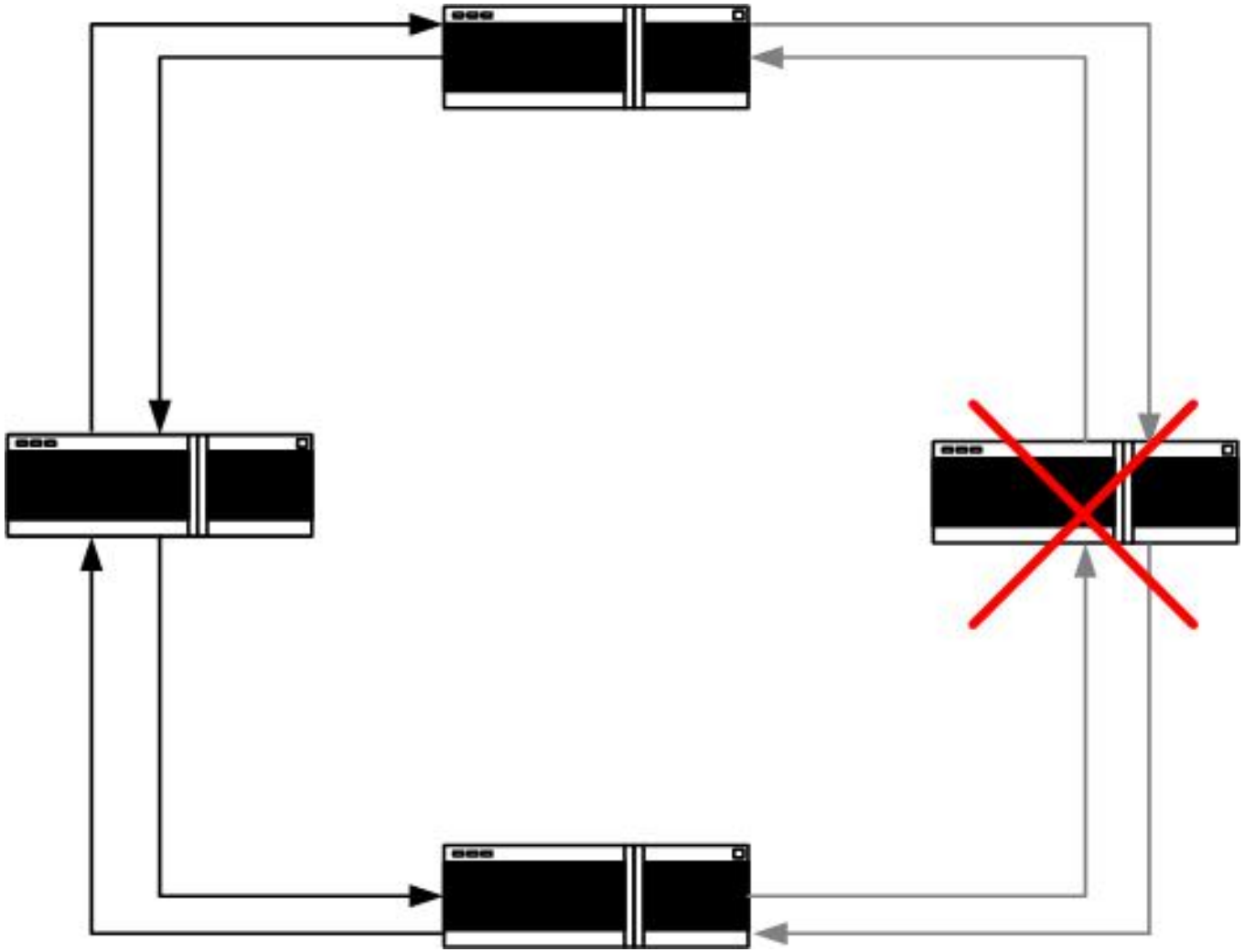
FDDI

- **Fiber Distributed Data Interface (FDDI)** ist eine Weiterentwicklung von Token Ring
- Medium: Glasfaserkabel
- doppelter gegenläufiger Ring (aktiver Ring, Reservering) mit Token-Mechanismus
- Datenrate: 100 MBit/s, 1000 MBit/s
- maximal 1000 Einheiten
- Ringlänge: max. 200 km
- maximaler Abstand zwischen zwei Einheiten: 2 km
- Fehlertoleranz (maximal eine Station)
- Nachrichten können hintereinander gelegt werden
- Weitere Entwicklungen FDDI-2

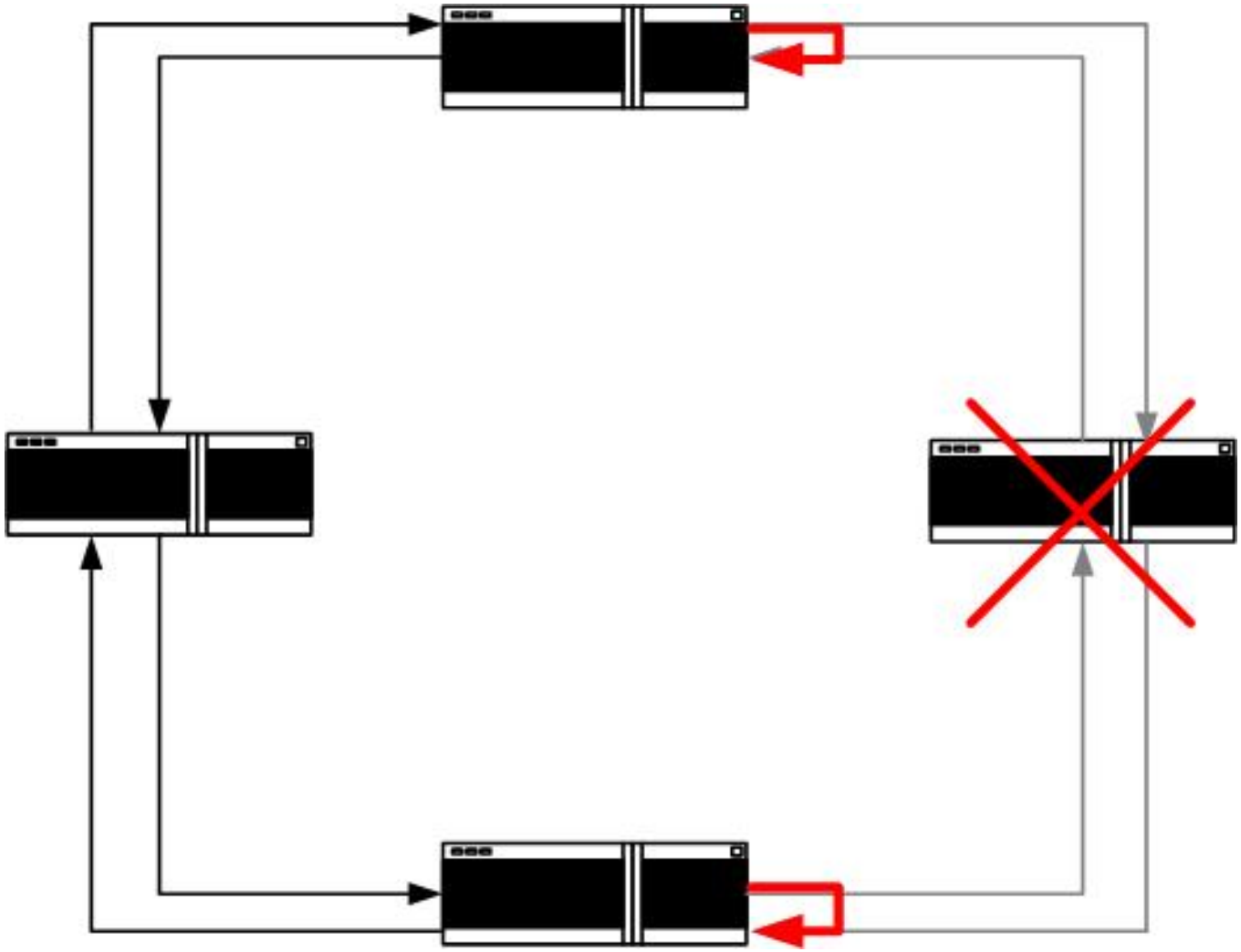
Fehlerkonfiguration FDDI



Fehlerkonfiguration FDDI



Fehlerkonfiguration FDDI



TDMA (Vertreter TTP/C)

TDMA

Mit **Time Division Multiple Access (TDMA)** werden Verfahren bezeichnet, die den Zugriff auf das Medium in Zeitabschnitte (slots) einteilen und jeweils einem Sender zu Verfügung stellen.

TDMA-Verfahren haben verschiedenste Vorteile:

- Kollisionen werden verhindert.
- Den einzelnen Sendern kann eine Bandbreite garantiert werden.
- Das zeitliche Verhalten ist deterministisch.
- Synchronisationsalgorithmen können direkt im Protokoll implementiert werden.

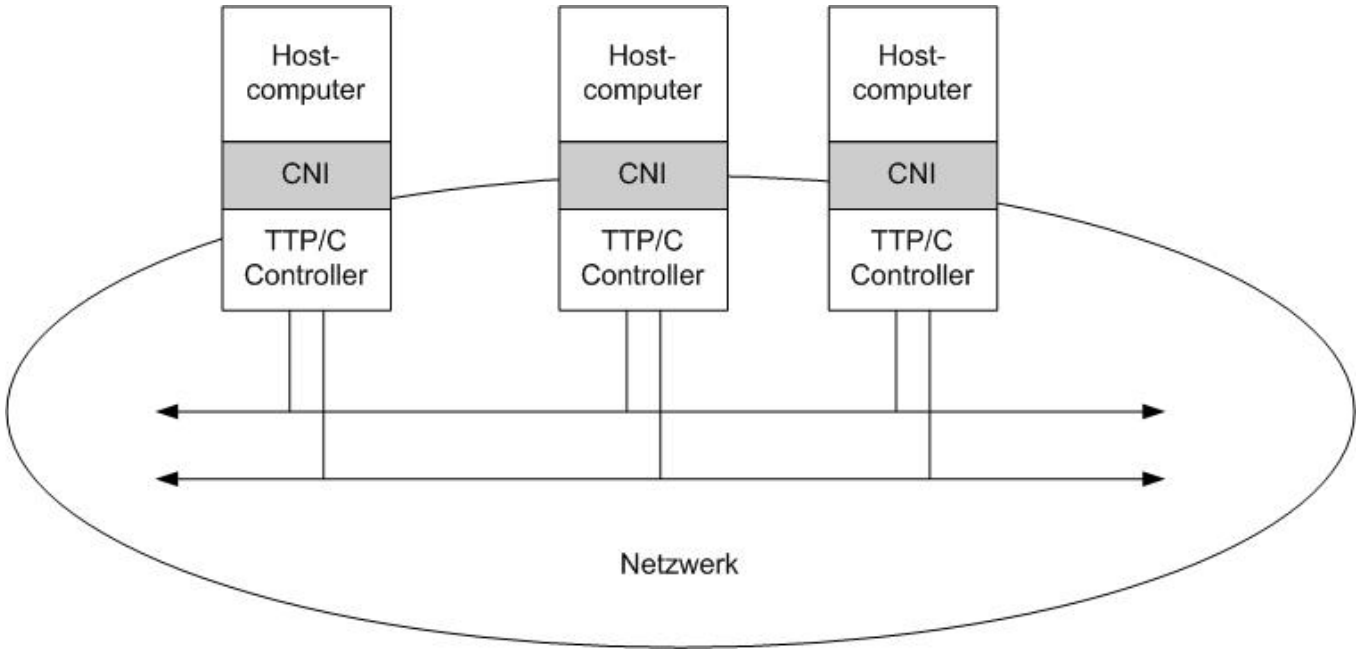
Nachteil:

- Bei reinem TDMA-Verfahren keine dynamische Zuteilung möglich.

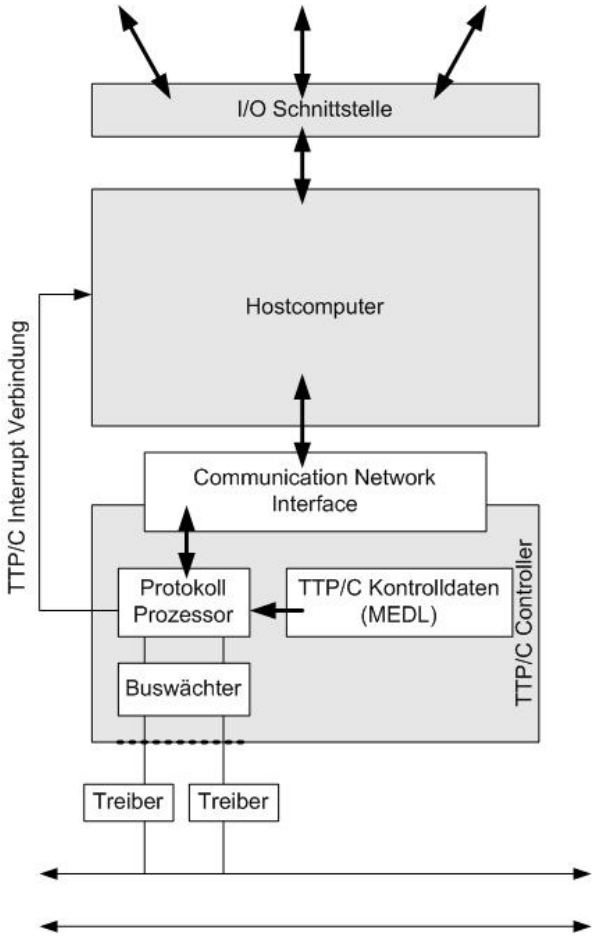
TTP/C

- Entstanden an der TU Wien, SpinOff TTTech
- TTP steht für Time Triggered Protocol
- TTP/C ist ein Netzwerksystem für harte Echtzeitsysteme.
- Ein verteilter, fehlertoleranter Uhrensynchronisationsalgorithmus (Einheit: 1 μs) toleriert beliebige Einzelfehler.
- Zwei redundante Kommunikationskanäle.
- Einheiten werden durch Guards geschützt (Vermeidung eines babbling idiots).
- Kommunikationsschema wird in Form einer MEDL (Message Descriptor List) a priori festgelegt und auf die Einheiten heruntergeladen.
- Einsatz unter anderem im Airbus A380

TTP-Architektur



TTP: Elektronisches Modul



Erläuterung

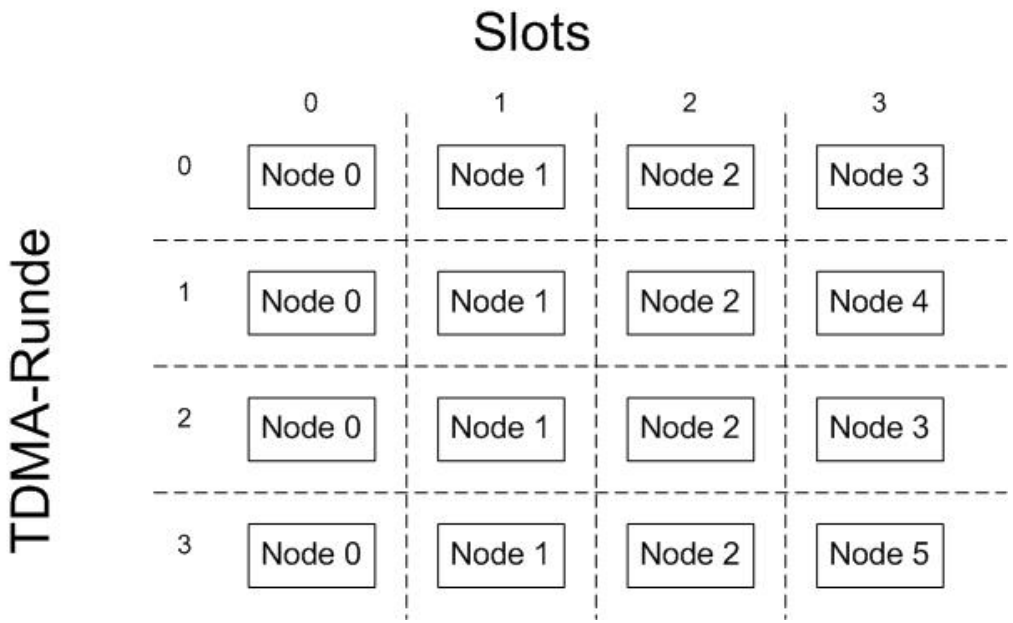
- Hostcomputer: Ausführung der eigentlichen Anwendung
- CNI: Gemeinsamer Speicherbereich von Hostcomputer und TTP/C-Kontroller
- Interruptverbindung: zur Übermittlung von Ticks der globalen Uhr und aussergewöhnlicher Ereignisse an den Hostcomputer
- MEDL: Speicherplatz für Kontrolldaten

Arbeitsprinzip

Der Kontroller arbeitet autonom ohne Kontrollsignale des Hostcomputers. Sämtliche notwendigen Kontrollinformationen sind in der MEDL enthalten:

- für jede zu empfangende und sendende Nachricht: Zeitpunkt und Speicherort in der CNI
- zusätzliche Informationen zur Ausführung des Protokolls

Beispiel: TDMA-Schema



Unterscheidung zwischen **reellen Knoten** (Knoten mit eigenem Sendeslot) und **virtuellen** Knoten (mehrere Knoten teilen sich einen Slot). Jeder reelle oder virtuelle Knoten sendet in einer TDMA-Runde genau einmal, die Länge der Sendeslots kann sich zwischen den Knoten unterscheiden, für einen Knoten ist sie aber immer gleich. $\Rightarrow$ gleich lang dauernde TDMA-Runden.

Protokolldienste

Das Protokoll bietet:

- Vorhersagbare und kleine, nach oben begrenzte Verzögerungen aller Nachrichten
- Zeitliche Kapselung der Subsysteme
- Schnelle Fehlerentdeckung beim Senden und Empfangen
- Implizite Nachrichtenbestätigung durch Gruppenkommunikation
- Unterstützung von Redundanz (Knoten, Kanäle) für fehlertolerante Systeme
- Unterstützung von Clustermoduswechseln
- Fehlertoleranter, verteilter Uhrensynchronisationsalgorithmus ohne zusätzliche Kosten
- Hohe Effizienz wegen kleinem Protokoll-Overhead

Fehlerhypothese

- Interne physikalische Fehler: Erkennung einerseits durch das Protokoll, sowie Verhinderung eine babbling idiots durch Guards.
- Externe physikalische Fehler: Durch redundante Kanäle können diese Fehler toleriert werden.
- Designfehler des TTP/C Kontrollers: Es wird von einem fehlerfreien Design ausgegangen.
- Designfehler Hostcomputer: Protokollablauf kann nicht beeinflusst werden, allerdings können inkorrekte Daten erzeugt werden.
- Permanente Slightly-Off-Specification-Fehler: können durch erweiterte Guards toleriert werden.
- Regionale Fehler (Zerstören der Netzwerkverbindungen eines Knotens): Folgen können durch Ring- und Sternarchitektur minimiert werden.

Medienzugriff

- Broadcast-Mechanismus mit TDMA-Schema
- Zum Versenden werden beide Kanäle benutzt (Daten müssen jedoch nicht redundant sein)
- Durch Einfügen eines Inter-Frame-Gaps (IFG) kann eine Überlappung der TDMA-Sendefenster vermieden werden
- Passive Knoten können mithören, aber keine Daten versenden.
- Schattenknoten sind passive redundante Knoten, die im Fehlerfall eine fehlerhafte Komponente ersetzen können.

Kontrollerzustand

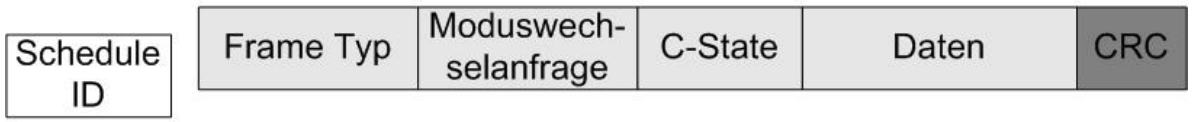
Der Kontrollerzustand (C-State) ist eine Menge von Zustandsvariablen, wird von dem Kontroller berechnet und muss mit den Zuständen der anderen Knoten übereinstimmen. Ein Knoten besitzt nur dann die Mitgliedschaft im TTP-Netzwerk, wenn sein Zustand mit der Mehrheit der anderen Knoten übereinstimmt.
Der Zustand enthält:

- die globale Zeit der nächsten Übertragung
- das aktuelle Fenster im Clusterzyklus
- den aktuellen, aktiven Clustermodus
- einen eventuell ausstehenden Moduswechsel
- den Status aller Knoten im Cluster

Sobald der lokale Zustand gültig ist, wird dies dem Hostcomputer mitgeteilt.

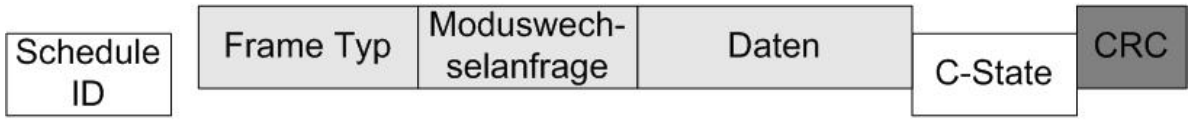
Frames in TTP/C

Frame mit explizitem C-State



Kaltstart-Frame

Frame mit impliziten C-State



Korrekte Knoten

- Ein Knoten ist dann korrekt, wenn er in seinem Fenster eine korrekte Nachricht versendet hat.
- Knoten können sich durch die Übernahme der Zeit und der Schedulingposition integrieren.
- Sobald ein sich integrierender Rechner in seinem Sendeslot eine korrekte Nachricht sendet, erkennen die anderen Rechner den Knoten an.

Start des Clusters

Der Start erfolgt in drei Schritten:

1. Initialisierung des Hostcomputers und des Controllers
2. Suche nach Frame mit expliziten C-State und Integration
3. Falls kein Frame empfangen wird, werden die Bedingungen für einen Kaltstart geprüft:
   - Host hat sein Lebenszeichen aktualisiert
   - Das Kaltstart Flag in der MEDL ist gesetzt
   - die maximale Anzahl der erlaubten Kaltstarts wurde noch nicht erreicht

   Sind die Bedingungen erfüllt, sendet der Knoten ein Kaltstartframe.

Sicherheitsdienste

- Korrektheit: Alle Knoten werden über die Korrektheit der anderen Knoten mit einer Verzögerung von etwa einer Runde informiert.
- Cliquenentdeckung: Es werden die Anzahl der übereinstimmenden und konträren Knoten gezählt. Falls mehr konträre Knoten gezählt werden, so wird ein Cliquenfehler angenommen.
- Host/Kontroller Lebenszeichen: der Hostcomputer muss seine Lebendigkeit dem Kontroller regelmäßig zeigen. Sonst wechselt der Kontroller in den passiven Zustand.

Uhrensynchronisation

- In regelmäßigen Abständen wird die Uhrensynchronisation durchgeführt.
- Es werden die Unterschiede der lokalen Uhr zu ausgewählten (stabilen) Uhren (mind.4) anderer Rechner anhand den Sendezeiten gemessen.
- Die beiden extremen Werte werden gestrichen und vom Rest der Mittelwert gebildet.
- Die Rechner einigen sich auf einen Zeitpunkt für die Uhrenkorrektur.